Theses and Dissertations   1. Thesis and Dissertation Collection, all items

1995-03

Distributed hard real-time scheduling for a software prototyping environment

Menezes Cordeiro, Mauricio de.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/31595

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

DISTRIBUTED HARD REAL-TIME SCHEDULING
FOR
A
SOFTWARE PROTOTYPING ENVIRONMENT

by

Mauricio de Menezes Cordeiro

March 1995

Dissertation Supervisor: Man-Tak Shing

19950814 070

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE (Leave Blank)	2. REPORT DATE March 1995	3. REPORT TYPE AND DATES COVERED Doctoral Dissertation

4. TITLE AND SUBTITLE DISTRIBUTED HARD REAL-TIME SCHEDULING FOR A SOFTWARE PROTOTYPING ENVIRONMENT	5. FUNDING NUMBERS
6. AUTHOR(S) CORDEIRO, MAURICIO de MENEZES	

7. PERFORMING ORGANIZATION NAME(S) AND ADRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

    Scheduling analysis is one of the most important activities in hard real-time systems development since the correctness of hard real-time systems depends not only on the logical results of computation, but also on the time at which the results are produced. This dissertation aimed at the development of both fundamental theory and software tools to support efficiently and reliably the scheduling of distributed hard real-time systems. The major work of this dissertation focuses on non-preemptive hard real-time scheduling, for periodic and sporadic task sets, although some of the results are also applicable to the preemptive case.

    Several theorems for checking the schedulability of non-preemptive task sets are developed. Previous results on necessary and sufficient conditions for scheduling non-preemptive task sets are extended to cover the case when the task deadlines can be smaller or equal to their periods. The concept of transient and cyclic schedules is introduced to overcome the weakness of the traditional methods, which restrict the construction of a cyclic schedule to a fixed interval of length equal to the least common multiple of the periods.

14. SUBJECT TERMS Real-time, Hard Real-time, Real-time Scheduling, Hard Real-time Scheduling, Scheduling, Static Scheduling, Distributed Scheduling, Non-preemptive, Synchronization, Distributed Systems, Prototyping	15. NUMBER OF PAGES 181
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

13. ABSTRACT (Continuation)

An algorithm for reducing the schedule length of periodic task sets is developed to further enhance the schedulability of the hard real-time systems. Preliminary study on randomly graphs shows that the algorithm do produce near-optimal solution.

To ease the problem of synchronization among tasks in distributed hard real-time systems, we introduce the Fundamental Synchronization Theorem and a novel model for designing distributed hard real-time systems without explicit synchronization, and develop an Ada95 software architecture to support such a model. The application of this theorem will allow us to treat each set of tasks allocated to a particular processor, as a totally independent set, if the tasks satisfy the conditions described in the theorem. This approach will greatly decrease the difficulties in scheduling large distributed real-time systems.

One of the necessary steps in distributed hard real-time scheduling is the allocation of tasks to different processors in the distributed system. Algorithms for task allocation which minimize the inter-module communication costs are developed and implemented.

Finally, a timing model for handling different time references in rapid prototyping systems is introduced, to support the reuse of real-time components.

DISTRIBUTED HARD REAL-TIME SCHEDULING
FOR A SOFTWARE PROTOTYPING ENVIRONMENT

by

MAURICIO DE MENEZES CORDEIRO
Commander, Brazilian Navy
B.S., Brazilian Naval Academy, 1976
M.S., Naval Postgraduate School, 1987

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1995

Author: ███████████████████████
/ Mauricio de Menezes Cordeiro

Approved by: ████████████████████
Man-Tak Shing
Associate Professor of Computer Science
Dissertation Supervisor

███████████████████████████████████████

Luqi
Associate Professor of Computer Science

Sherif Michael
Associate Professor of Electrical and
Computer Engineering

███████████████   ███████████████

James Vincent Sanders
Associate Professor of Physics

Amr M. Zaky
Assistant Professor of Computer Science

Ted Lewis, Chairman, Department of Computer Science

Approved by: ████████████████
Richard S. Elster, Dean of Instruction

iii

ABSTRACT

Scheduling analysis is one of the most important activities in hard real-time systems development since the correctness of hard real-time systems depends not only on the logical results of computation, but also on the time at which the results are produced. This dissertation aimed at the development of both fundamental theory and software tools to support efficiently and reliably the scheduling of distributed hard real-time systems. The major work of this dissertation focuses on non-preemptive hard real-time scheduling, for periodic and sporadic task sets, although some of the results are also applicable to the preemptive case.

Several theorems for checking the schedulability of non-preemptive task sets are developed. Previous results on necessary and sufficient conditions for scheduling non-preemptive task sets are extended to cover the case when the task deadlines can be smaller or equal to their periods. The concept of transient and cyclic schedules is introduced to overcome the weakness of the traditional methods, which restrict the construction of a cyclic schedule to a fixed interval of length equal to the least common multiple of the periods. An algorithm for reducing the schedule length of periodic task sets is developed to further enhance the schedulability of the hard real-time systems. Preliminary study on randomly graphs shows that the algorithm do produce near-optimal solution.

To ease the problem of synchronization among tasks in distributed hard real-time systems, we introduce the Fundamental Synchronization Theorem and a novel model for designing distributed hard real-time systems without explicit synchronization, and develop an Ada95 software architecture to support such a model. The application of this theorem will allow us to treat each set of tasks allocated to a particular processor, as a totally independent set, if the tasks satisfy the conditions described in the theorem. This approach will greatly decrease the difficulties in scheduling large distributed real-time systems.

One of the necessary steps in distributed hard real-time scheduling is the allocation of tasks to different processors in the distributed system. Algorithms for task allocation which minimize the inter-module communication costs are developed and implemented.

Finally, a timing model for handling different time references in rapid prototyping systems is introduced, to support the reuse of real-time components.

TABLE OF CONTENTS

x

TABLE OF FIGURES AND TABLES

ACKNOWLEDGMENTS

I. INTRODUCTION TO HARD REAL-TIME SYSTEMS

A. INTRODUCTION

Traditionally, most real-time systems have been built for military purposes. As computers become faster, more inexpensive, and more reliable, a tendency towards automation is emerging in virtually every field of activity. Areas in which real time systems are being more widely employed include manufacturing, communications, defense, transportation, aerospace, energy, and health care.

"Hard real-time systems" are defined as those systems in which the correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced. They are also characterized by the fact that severe consequences will result if logical as well as timing correctness properties of the system are not satisfied. [SR88]

To put it briefly, real-time systems differ from traditional systems in that deadlines or other explicit timing constraints are attached to the tasks or processes.

Audsley and Burns presented a very interesting approach [AB93], where the time taken to complete a task is mapped against the value this task has to the system, developing the so called *time-value functions*. This work proposes an adaptation of their approach to be used by CAPS, where the time critical tasks could have several kinds of deadlines, as shown in Figure 1.1. Tasks with hard deadlines may cause damage to the system if they start early or finish late. Tasks with soft deadlines convey the main idea of "better late than never", and the tasks with hybrid deadlines can be assumed to have a soft deadline behavior until certain point in time, but then they become hard deadline tasks, generating damage to the system. Using this approach, it is possible to determine whether it is more convenient to preempt a task that has not finished within its deadline or keep it running. This approach provides a much better representation for a task deadline, than that achieved by merely calling it a soft or a hard deadline.

1

In general it can be said that there are three types of tasks, depending upon their deadline characteristics. The periodic tasks that execute on a regular basis, and usually have a period and a required execution time. The aperiodic tasks (also known as non-periodic) which are essentially random tasks triggered by some external event. Aperiodic tasks may also have some timing constraints that limit their maximum start or finish time. However, if aperiodic tasks are allowed to have hard deadlines (in other words, if they are allowed to have negative values once the deadline is missed) worst case analysis cannot be further discussed without further restricting their timing behavior. This is the rationale behind the third type of task, the sporadic task, in which a minimum period between any two aperiodic events is required. [AB93]
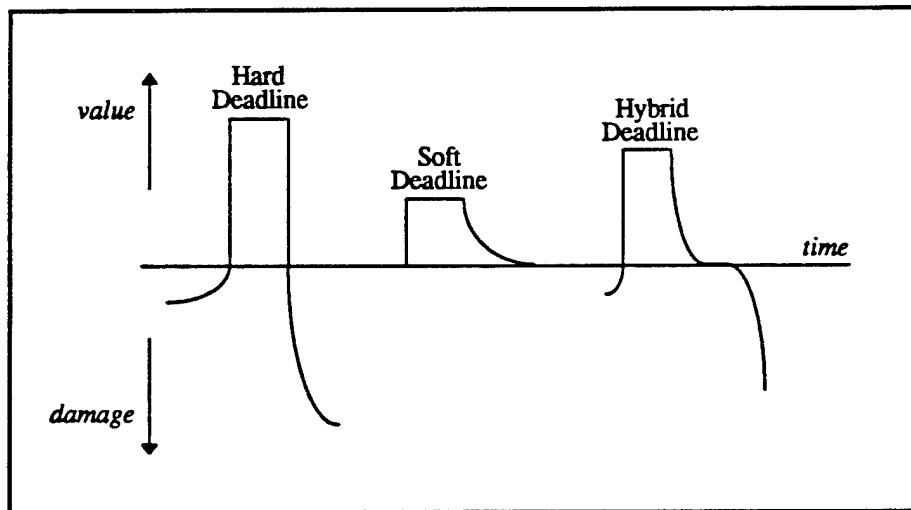
Figure 1.1. Types of Task Deadlines

In addition to timing constraints, a task can have other constraints, such as [SR88]:

1) *resource constraints* - which note the resources required during the execution of the task

2) *precedence constraints* - that specify a partial (perhaps total) ordering on the execution of the tasks

3) *concurrency constraints* - that describe which tasks can run concurrently, to share, for example, a resource

2

4) *placement constraints* - which note whether a given task is to run in a specific processor

5) *criticalness* - which is the relative value to the system that is associated with some specific task when it meets its deadline

6) *preemptiveness* - determining whether a task can be interrupted by other tasks and resume execution afterwards

7) *communication requirements* - that note issues, such as acceptable delays, for inter-task communications and synchronization protocols

Task scheduling in hard real-time systems can be either static or dynamic. In static scheduling it is assumed that all information about the tasks is known *a priori*, and the schedule is usually generated off-line. In dynamic scheduling, although all information about the tasks may be known *a priori*, they are allowed to be dynamically invoked, and the schedule is calculated "on the fly". There has been a great deal of debate about the appropriateness of dynamic algorithms for hard real-time systems. Many people are in favor of static scheduling because it seems reasonable to assume that for safety-critical applications all the schedulability should be guaranteed before execution [AB93].

B. REVIEW OF PREVIOUS WORK

According to Baker [Bak74], scheduling is the allocation of resources over time to perform a collection of tasks. This rather general definition conveys the basic idea of scheduling theory, which is a collection of principles, models, techniques and logical conclusions that provide insight into the scheduling function.

Many of the early developments in the field of scheduling were motivated by problems arising in manufacturing. Today, even though scheduling is used in many different areas, there are still references that deal with machines instead of processors, and with jobs instead of tasks.

In order to have a better understanding of the context in which scheduling issues are found, it is reasonable to begin by proposing a taxonomy for the scheduling function.

3

This taxonomy is an enhancement of that proposed by Cheng, et al. [CSR87] and is illustrated in Figure 1.2.

As shown in the figure, classical scheduling can be divided into four major areas: single-machine problems, parallel-machine, flow shop, and job shop scheduling. Most of these areas make use of objective functions, such as minimizing flowtime, minimizing mean tardiness, and minimizing completion time (makespan), which does not convey much of the important information needed by real-time systems. In most of these problem areas, the deadline concept is not even considered. Nevertheless, some of these results can provide very fruitful insights into real-time scheduling problems. Another issue that is not considered in many of the problems associated with classical scheduling is the idea of periodic tasks, meaning tasks that run forever. For further reading on classical scheduling the reader is directed to the work of Baker [Bak74] and Stankovic, et al. [SSN93]. The latter reference presents a concise survey on the implications of classical scheduling results for real-time systems.
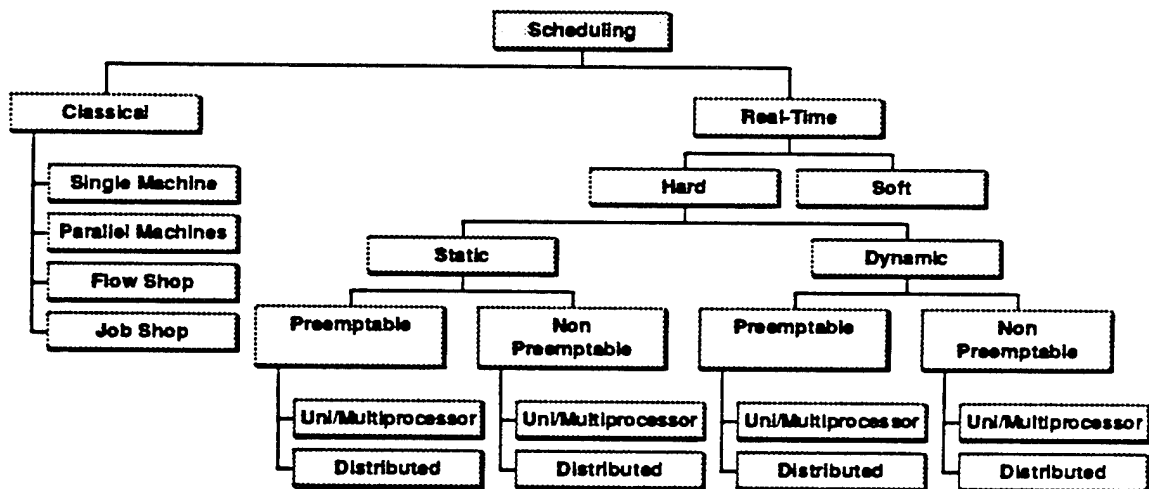
Figure 1.2. Scheduling Taxonomy

Tasks can also be distinguished as preemptable or non-preemptable. A task is preemptable if it can be interrupted by other tasks and can resume execution afterwards. A non-preemptable task, once started, must run to completion.

Another concept that requires introduction is the difference between multiprocessor systems and distributed systems. In multiprocessor systems, the cost of interprocessor communications is negligible, as the different processors usually have some kind of shared memory and a global clock. In distributed systems, the cost of interprocessor communications is not negligible, as the processors do not share any memory space and each processor has its own clock. It is now appropriate to make a brief review of some previous work done in hard real-time scheduling, with an emphasis on the results related to static scheduling.

1. Preemptive Static Scheduling

In cases where the tasks are periodic, which is the most common case in real-time systems, it can be said that the most important result for the uniprocessor case was provided by Liu and Layland [LL73]. They proved that the Earliest Deadline First (EDF) algorithm is optimal for any set of independent periodic tasks where optimality is defined by the statement, "if a set of tasks can be scheduled by any algorithm, then it can be scheduled by the EDF algorithm". They also demonstrated some bounds on processor utilization when using this algorithm. Their results were extended to cover cases where the release times are arbitrary by Jeffay [Jef89a]. Also based on Liu and Layland's work, a more elaborate schedulability test was proposed by Lehoczky, et al. [LSD89]. This test employed the concept of processor time demand for handling cases where the deadlines were smaller than the periods. Sha and Lehoczky [LS86] described a technique of splitting the periods so that better processor utilization could be achieved.

Horn [Hor74] developed an optimal $O(n^2)$ algorithm that was also based on the earliest deadline first principle. Originally formulated for non-periodic tasks, this algorithm proved capable of handling independent tasks with arbitrary deadlines and release times in a uniprocessor environment. For the same type of tasks, he also introduced an algorithm for the multiprocessor case that was based on the network flow method. Martel [Mar82] extended the work of Horn by allowing for processors with different speeds.

5

For multiprocessor scheduling of periodic tasks, most of researchers have adopted a partition approach, where some kind of bin-packing algorithm is used to determine the sub-optimal partitions. Examples can be found in the work of Davari and Dhall [DD86], Bannister and Trivedi [BT83], and in that of Dhall and Liu [DL78].

2. Non-Preemptive Static Scheduling

There has been a great deal of research in the area of preemptive real-time scheduling. For the non-preemptive case, however, most problems have been shown to be NP-hard, even in the uniprocessor case. Hence, the majority of the work that has been done in this area covers very specific cases, such as when unit computation times are involved, or when release times are the same. Moore [Moo68] showed that the earliest deadline algorithm is optimal for scheduling a set of independent tasks that have the same release time. Bratley, Florian and Robillard [BFR71] developed an implicit enumeration algorithm to determine scheduling for non-preemptive tasks with arbitrary release times and deadlines. Baker and Su [BS74] used a similar approach to minimize the maximum tardiness of tasks. Erschler, et al. [EFM83] developed a necessary condition for scheduling tasks with arbitrary release times and deadlines. When utilizing periodic task sets, which are definitely the major area of focus for this study, the major results can be found in the work of Mok [Mok83], Xu [XP90], Jeffay [JSM91] and Zhu [ZLC94].

3. Summary of Scheduling Complexity

In dealing with scheduling problems where most of the input instances have been proven to be NP-hard, it is very important and beneficial to know in which class a particular instance belongs, so that the problem can be addressed appropriately. However, when one looks into the huge amount of research in this area, it becomes apparent that the various studies are very difficult to compare. While it is undesirable to limit the creativity of researchers, it is increasingly apparent that some kind of standard is needed, so that individual research efforts at least speak in the same language.

Nevertheless, this section offers a summary of the major results achieved in the area of time complexity of scheduling algorithms, for both the preemptive and non-preemptive cases. Whenever the result is applicable to periodic task sets, it will be briefly mentioned.

In Table 1.1, it has been listed, for each case, the number of processors (m), the precedence relation (<) among the tasks (if one exists), the valid domain for the release time (r_i), the deadline (d_i), the computation time (c_i), whether it is preemptive or non-preemptive, the time complexity of the problem, the reference paper, and, finally, some additional remarks. Note that in this table most of the results are for non-periodic task sets. In the following section, the problem of how to apply these results to the periodic case is addressed.

Preemptive

m	Preced. Relations	r_i	d_i	c_i	Complexity	Reference	Remark
arb	arbitrary	0	arb	arb	NPC	GJ77a	
arb	forest	k	∞	arb	O(nlogm)	GJ77b	
arb	tree	0	∞	arb	$O(n^2)$	MC70	
arb	tree	0	∞	arb	O(nlogm)	GJ77b	
arb	empty	arb	∞	arb	$O(n^3)$	Hor74	Network Flow Same Speed Processors
arb	empty	arb	∞	arb	$O(m^2n^4+n^5)$	Mar82	Network Flow Different Processors
arb	empty	arb	∞	arb	O(nlogn)	DD86	EDF ($d_i = p_i$)or RM + O(n) Next-Fit
1	arbitrary	arb	arb	arb	$O(n^2)$	Bla76	EDF based
1	empty	arb	∞	arb	$O(n^2)$	Hor74	EDF based
1	empty	0	arb	arb	$O(n^2)$	LL73	rate-monotonic periodic tasks ($d_i = p_i$)

Non-Preemptive

arb	tree	0	∞	1	O(nlogn)	Hu61	
arb	empty	arb	arb	1	$O(n^3loglogn)$	Sim83	Barrier's Alg.
2	arbitrary	arb	arb	1	$O(n^3)$	GJ77a	
1	arbitrary	0	∞	arb	$O(n^2)$	Law73	Backward EDF
1	arbitrary	arb	arb	1	O(nlogn)	GJS81	Forbidden Regions Alg. (Very complex data structures)
1	empty	arb	arb	1	O(nlogn)	Jac55	EDF (Minimizes Completion Time)
1	empty	0	arb	arb	$O(n^2)$	Moo68	EDF

Table 1.1. Major Results in Scheduling Algorithms

Table 1.2 summarizes the complexity boundaries of various non-preemptive problems with respect to the number of processors, computation time, and type of partial order.

m	Preced. Relations	r_i	d_i	c_i	Complexity	Reference
$k \geq m > 2$	arbitrary	0	k	1	OPEN	
arb	arbitrary	0	k	1	NPC	Ull75
arb	empty	0	k	arb	NPC	Ull75
2	arbitrary	0	k	1,2	NPC	Ull75
2	empty	0	k	arb	NPC	Ull75
1	empty	arb	arb	arb	NP-hard	GJ77a
$k \leq m \leq 2$	arbitrary	0	k	1	P	CG72
arb	tree	0	k	1	P	Hu61
arb	empty	0	k	1	P	Ull75

Table 1.2. Summary of Non-Preemptive Scheduling Complexity

Table 1.3 is very interesting in the sense that it delimits the boundaries between NP-completeness and polynomial solvability for the more constrained non-preemptive scheduling problem, where resources (Rsrc) other than processors are being requested by the tasks. As can be seen, by having no precedence relations, or for values of m less than 2 in the first case, or by making m less than three in the second case, the resulting problems can be solved in polynomial time. [GJ75]

m	Preced. Relations	r_i	d_i	c_i	Complexity	Reference	Remark
$m \geq 2$	forest	0	k	1	NPC	GJ75	Rsrc ≥ 1
$m \geq 3$	empty	0	k	1	NPC	GJ75	Rsrc ≥ 1

Table 1.3. Complexity of the Scheduling Problem with Several Resources

Other important results are:

"It is impossible to find a totally optimal run-time scheduler even if any ready process is permitted to preempt any other process in progress".[Mok76]

"When there are mutual exclusion constraints, it is impossible to find a totally on-line optimal run-time scheduler".[Mok83]

"The problem of deciding whether it is possible to schedule a set of periodic processes which use semaphores only to enforce mutual exclusion is NP-hard".[Mok83]

"The problem of computing a static schedule for a set of periodic timing constraints is NP-hard".[Mok83]

"Non-preemptive scheduling of periodic tasks when release times are taken into consideration is NP-hard in the strong sense".[JSM91]

"The processor allocation problem is NP-complete even for the case where only two processors are available and the processor scheduling problem resulting from any partition is easy".[Mok83]

"The problem of finding an optimal schedule is NP-hard for a single processor even if all tasks have the same ready time and deadline".[LW90]

4.   A Brief Note about the Periodic Task Complexity

It is very common for authors of papers that deal with the scheduling of non-periodic tasks, i.e., tasks that are executed only once, to infer that their algorithms or methods can also be applicable to periodic tasks by simply applying the same algorithm to the set of tasks occurring within a time period that is equal to the least common multiple of their periods.

Although this assertion is true in most of cases, one must note that a polynomial time algorithm for scheduling non-periodic tasks may take exponential time to schedule a set of periodic tasks using the same algorithm. To see this, consider an algorithm A that schedules a set T of n non-periodic tasks in time $O(|I|^3)$, where $|I|$ is equal to the size of the input instance. Clearly, by using a binary encoding, $O(n + \Sigma \log r_i + \Sigma \log c_i + \Sigma \log d_i)$ bits are needed to encode such an instance. Now, assume a set T' of n periodic tasks with periods $p_1, p_2, ... , p_n$, whose input size is $O(n + \Sigma \log r_i + \Sigma \log c_i + \Sigma \log d_i + \Sigma \log p_i)$. Note that in the worst case an LCM of $p_1 \times p_2 \times ... \times p_n$ exists. So, in order to use algorithm A to schedule the periodic task set T', one must first transform T' into an equivalent set T'' of non-periodic tasks with $p_2 \times p_3 ... \times p_n$ instances of task T_1, $p_1 \times p_3 ... \times p_n$ instances of task T_2, $p_1 \times p_2 ... \times p_n$ instances of task T_3, and so on.

Clearly, the size $|T''|$ of the input instance T'' is equal to

$$O(\; n + \sum_{i=1}^{n} \; [(\; \log r_i + \log c_i + \log d_i) \times \frac{p_1 \times p_2 \times .. \times p_n}{p_i}\;]\;),$$

and algorithm A will take $O(|T''|^3)$ time to schedule all task instances in T''. But, since $|T''|$

$\nleq C \times (\; [\; n + \sum_{i=1}^{n} \; (\; \log r_i + \log c_i + \log d_i + \log p_i\;)\;]^k\;)$ for any constants C and k, $O(|T''|^3)$

is exponential with respect to $|T'|$.

5. Complexity Results for Message Routing in Distributed Systems

This section presents some very interesting results from Leung [LTW89] regarding the possibility or impossibility of sending a set of messages in a distributed real-time system on-time. Each message M is represented by the quintuple $(s_i, e_i, l_i, r_i, d_i)$ where s_i denotes the origin node for M_i, e_i denotes the destination node, l_i is the length of M_i, r_i is the release time, and d_i denotes the deadline of M_i. The problem was studied for both preemptive and non-preemptive cases, but this discussion will be restricted to the latter. It is also assumed that the processors are connected by an uni-directional ring. Table 1.4 shows the complexity results for the non-preemptive transmission. An entry marked k denotes that the parameter is the same for all messages, while a V entry denotes that it can vary according to the message.

s_i	e_i	r_i	d_i	Complexity
V	k	k	k	P
k	V	k	k	P
k	k	V	k	P
k	k	k	V	P
k	k	V	V	NP
k	V	k	V	NP
k	V	V	k	NP
V	k	k	V	NP
V	k	V	k	NP
V	V	k	k	NP

Table 1.4. Complexity for Non-Preemptive Transmissions

As shown in Table 1.4, the message routing problem becomes NP whenever two or more parameters are allowed to be arbitrary. These and other results had a great influence on the manner in which this dissertation will treat distributed scheduling.

II.   CAPS AND PSDL OVERVIEW

A.   MOTIVATION

The United States Department of Defense (DoD) is currently the world's largest user of computers. Each year, billions of dollars are allocated for the development and maintenance of progressively more complex weapons and communications, and information systems. These systems increasingly rely on information processing, utilizing embedded computer systems, and are often characterized by time periods or deadlines within which some event must occur. Such periods or deadlines are known as "hard real-time constraints". Satellite control systems, missile guidance systems, and communications networks are examples of embedded systems with hard real-time constraints. The correctness and reliability of these software systems is critical, making software development of these systems an immense task with increasingly high costs and potential for design errors [Boo87].

Over the past twenty years, technological advances in computer hardware technology have reduced the hardware portion of total system cost from 85 percent to about 15 percent. In the early 1970s, studies showed that computer software alone comprised approximately 46 percent of the total estimated DoD computer costs. Of this cost, 56 percent was devoted specifically to embedded systems. In spite of the tremendous expense, most large software systems were characterized as not providing the functionality that was desired, taking too long to develop, costing too much time or taking too much space to use, and lacking the ability to evolve to meet the user's changing needs [Boo87].

Software engineering evolved in response to the need to more efficiently design, implement, test, install, and maintain larger and more complex software systems. The term "software engineering" was coined in 1967 by a NATO study group, and endorsed by the 1968 NATO Software Engineering Conference [Sch90]. The conference concluded that software engineering should use the philosophies and paradigms of

traditional engineering disciplines. Numerous methodologies have been introduced to support software engineering. The major approaches which underlie these different methodologies are the waterfall model [Lam88], the spiral model [Boe86], and the prototyping methods of development [Luq89].

B.   THE WATERFALL MODEL

The waterfall model describes a sequential approach to software development as shown in Figure 2.1. The requirements are completely determined before the system is designed, implemented and tested. The cost of systems developed using this model is very high. Required modifications that are realized late in the development of a system, such as during the testing phase, have a much greater impact on the cost of the system than they would have if they had been determined during the requirements analysis stage of development. Requirements analysis may be considered the most critical stage of software development, since this is when the system is defined.
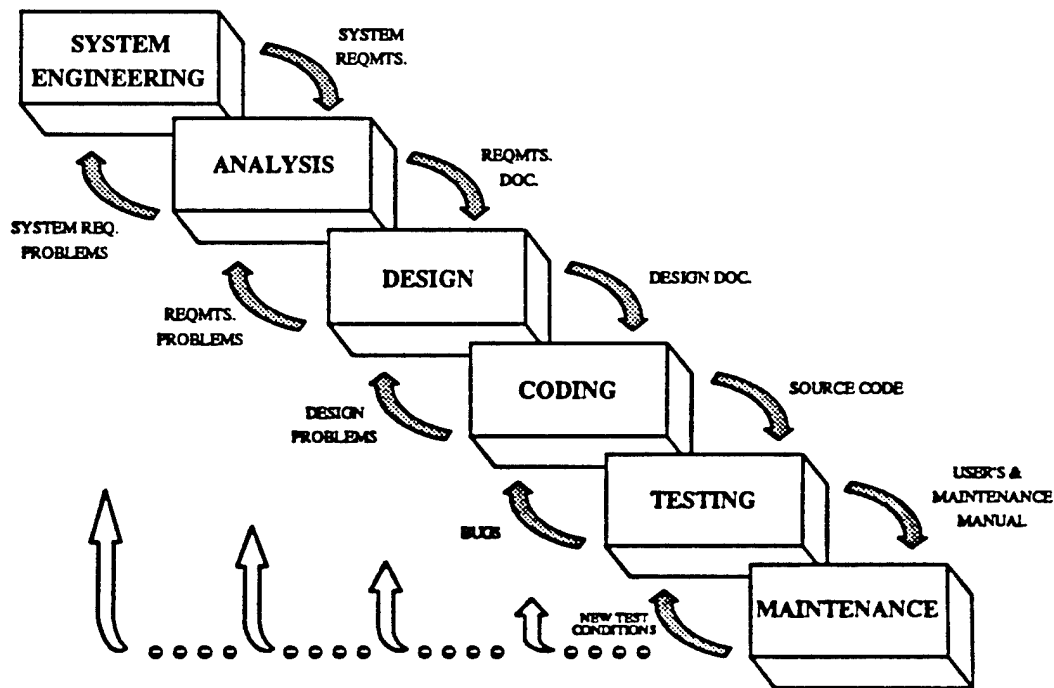
Figure 2.1. The Waterfall Model

Requirements are often incompletely or erroneously specified, due to the often vast difference in the technical backgrounds of the user and the analyst. It is often the case that the user understands his application area but does not have the technical background to communicate his needs to the analyst, while the analyst is not familiar enough with the application to detect a misunderstanding between himself and the user. The successful development of a software system is strictly dependent upon this process. The analyst must understand the needs and desires of the user and the performance constraints of the intended software system in order to specify a complete and correct software system.

Requirements specifications are still most widely written using the English language, which is an ambiguous and non-specific mode of communication.

Another difficulty of the classical life cycle is that communication between a software development team and the customer or the system's users is weak. Most of the time the customer does not know what he or she wants. In that case it is hard to determine the exact requirements, since the software developer is also unfamiliar with the problem domain of the system. Formal specification languages are used to formalize customer needs to a certain extent. Another disadvantage of the classical project life cycle is that a working model of the software system is not available until late in the project time span. This may cause two things:

1) A major bug that remains undetected until the working program is reviewed, which can be disastrous [Pre87];

2) The customer will not a have an idea of what the system will look like until it is complete.

C. THE SPIRAL MODEL

Large real-time systems and systems which have hard real-time constraints are not well supported by traditional software development methods because the designer of this type of system would not know if the system can be built with the timing and control constraints required until after much time and effort has been spent on implementation. A

15

hard real-time constraint imposes a time-bound on the response time of a process which must be satisfied under all operating conditions.

To solve the problems raised in requirements analysis for large, parallel, distributed, real-time, or knowledge-based systems, current research suggests an alternative paradigm for software development and evolution based on rapid prototyping [LB88]. The purpose of prototyping is to ensure that proposed requirements and system concepts adequately match the needs of the prospective client(s) before detailed optimization and implementation efforts begin. As a software methodology, rapid prototyping provides the user with increasingly refined systems to test and the designer with ever better user feedback between each refinement. The result is more user involvement throughout the development/specification process, and consequently, better engineered software.

The prototyping method shown in Figure 2.2 has recently become popular. "It is a method for extracting, presenting, and refining a user's needs by building a working model of the ultimate system – quickly and in context" [Boa84]. This approach captures an initial set of needs, and quickly implements those needs with the stated intent of iteratively expanding and refining them as the user's and designer's understanding of the system grows. The prototype is only to be used to model the system's requirements, rather than as an operational system [You89].
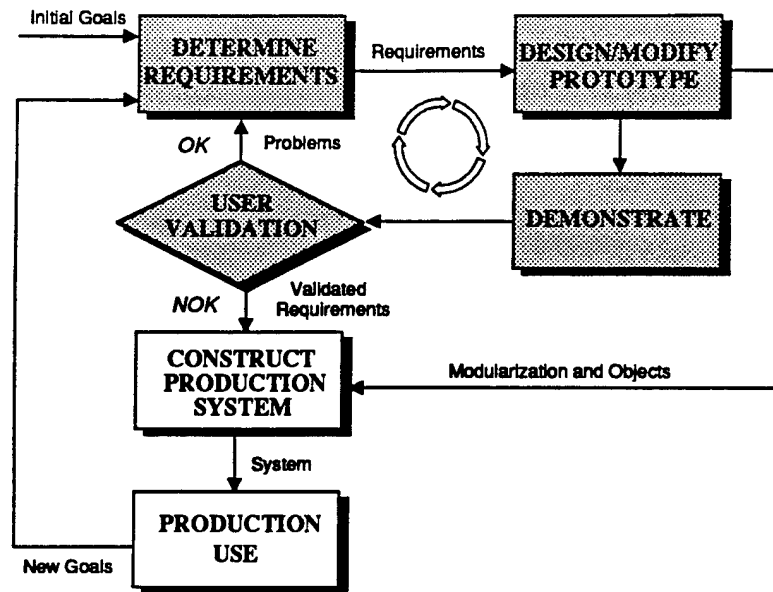
Figure 2.2. The Prototyping Process

This iterative prototyping process is also known as the "Spiral Model of Software Development" and is illustrated in Figure 2.3. In the prototyping cycle, the system designer and the user work together at the beginning of the project to determine the critical parts of the proposed system. The designer then implements a prototype of the system based on these critical requirements by using a prototype description language [Luq89]. The resulting system is presented to the user for evaluation. During these demonstrations, the user determines whether the prototype behaves as it is supposed to do, examines user interface options, and, most importantly, verifies understanding of the problem and solution. If errors are found at this point, the user and the designer work together again on the specified requirements to correct them. Concurrently, a risk analysis is initiated to decide whether or not to move on to the next cycle of the spiral. This process continues until the user determines that the prototype successfully captures the critical aspects of the proposed system. This is the point where precision and accuracy are obtained for the proposed system. The designer then uses the prototype as a basis for designing the production software.

17
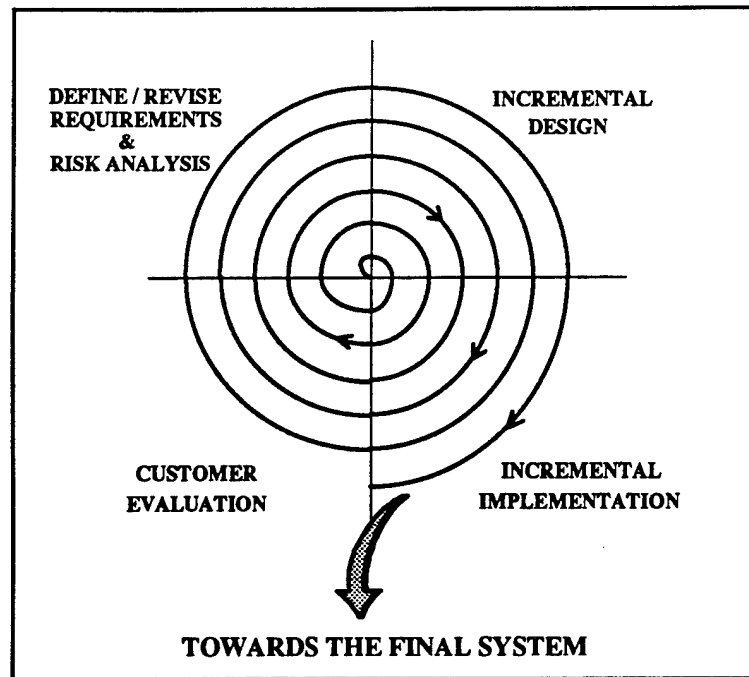
Figure 2.3. The Spiral Model

Some advantages and disadvantages of iterative development methodology are listed below:

Advantages:

1) There is constant customer involvement (revising requirements).

2) Software development time is greatly reduced.

3) Methodology maps to reality.

4) It allows use of off-the-shelf tools.

Disadvantages:

1) There are configuration control complexities.

2) The developer is compelled to manage customer enthusiasm.

3) There are uncertainties in contracting the iterative development.

Manually construction of the prototype still takes too much time, and can introduce many errors. Also, it may not accurately reflect the timing constraints placed upon the system. What is needed is an automated method of rapidly prototyping a hard

real-time system that reflects those constraints and requires minimal development time. Such a system should exploit reusable components and validate timing constraints.

If Ada software that is reliable, affordable, and adaptable is to be produced and maintained, the characteristics of Ada may not be the only important matter to consider, as the characteristics of Ada software development environments may well be critical [BL91].

The rapid, iterative construction of prototypes within a computer aided environment automates the prototyping method of software development, and is called rapid prototyping. Rapid prototyping provides an efficient and precise means to determine the requirements for the software system, and greatly improves the likelihood that the software system developed from the requirements will be complete, correct, and satisfactory to the user. The potential benefits of prototyping depend critically on the ability to modify the behavior of the prototype with less effort than that required to modify the production software. Computer aided and object-based rapid prototyping provides a solution to this problem.

D.   THE COMPUTER AIDED PROTOTYPING SYSTEM (CAPS)

The Computer-Aided Prototyping System (CAPS) [LK88] is a software engineering tool for developing prototypes of real-time systems. It is useful for requirements analysis, feasibility studies, and the design of large embedded systems. CAPS is based on the Prototype System Description Language (PSDL) [LBY88], which provides facilities for modeling timing and control constraints within a software system. An overview of PSDL will be presented in the following section. CAPS is a development environment, implemented in the form of an integrated collection of tools, linked together by a user-interface, and provides the following kinds of support to the prototype designer:

- timing feasibility checking via the scheduler,
- consistency checking and some automated assistance for project planning, scheduling, designer task assignment, and project completion date estimation via the Evolution Control System,

- design completion via the editors,

- computer-aided software reuse via the software base.

A CAPS prototype is initially built as an augmented data flow diagram and a corresponding PSDL program. The CAPS data flow diagram and PSDL program are augmented with timing and control constraint information, which is used to model the functional and real-time aspects of the prototype. The CAPS environment provides all of the necessary tools for engineers to quickly develop, analyze, and refine real-time software systems.

The general structure of CAPS is shown in Figure 2.4. The CAPS User-Interface provides access to all of the CAPS tools, and facilitates communication between tools when necessary. The tools in Figure 2.4 are grouped into four sections: Editors, Execution Support, Project Control and Software Base. Each CAPS tool is associated with a different aspect of the CAPS prototyping process.
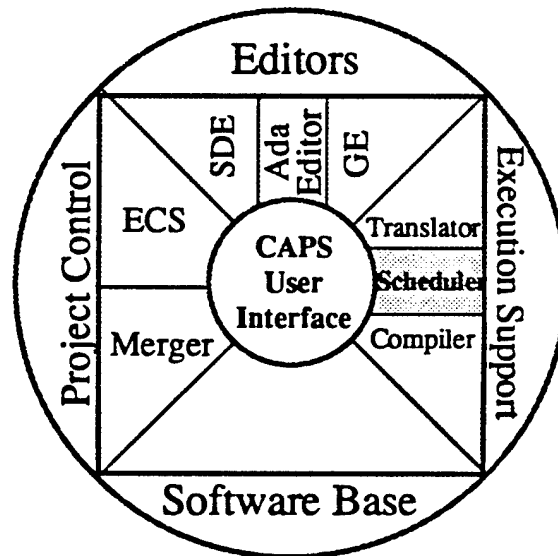
Figure 2.4. The CAPS Structure (from Bro[94])

CAPS is specifically designed to assist and partially automate development efforts which lie in the shaded regions of the prototyping process (Figure 2.2). Specifically, based

on a set of initial requirements, CAPS allows the engineer to design, modify, demonstrate and validate a software system. Through this process, system requirements can be refined and modified as necessary.

The CAPS prototyping process is more specific, and it could be said that it is a refinement of what is shown in Figure 2.2, and is outlined below. [Bro94]

1) Based on requirements, design (or modify) the data flow diagram for the system

2) Assign all appropriate timing and control constraints to the prototype operators. Assign latencies to data streams (if required)

3) Assign data types to all data streams

4) Find (in the software base) or build an implementation module for each user-defined data type and each atomic operator. Modules taken from the software base can be modified after retrieval to suit individual needs

5) Build the prototype's user-interface (if required)

6) Translate the CAPS-generated (and user-augmented) PSDL program into (a portion of) the Ada supervisor module

7) Run the CAPS scheduler to generate the static and dynamic schedules. This completes the prototype's Ada supervisor module

8) Compile the prototype. (Note: for successful compilation, particular attention must be paid to the formal parameters of atomic operator implementation procedures created in step 4)

9) Execute, evaluate and modify (if appropriate) the prototype and/or the requirements

10) Return to Step 1 if prototype modification is required

The correlation between these 10 steps and Figure 2.2 is obvious. Note that the basic 10 steps are a bit more detailed than the preceding prototyping process diagram. This highlights the real-time requirements, and associated design considerations of typical CAPS prototypes.

The remainder of this introduction briefly introduces the CAPS tools used to perform the basic 10 steps. Note, also, that two of the CAPS tools are outside the purview of the prototyping process diagram. These tools perform ancillary functions which are not seen in either the prototyping process diagram or the 10 basic CAPS steps. These advanced feature tools are the Evolution Control System and the Merger.

The purpose of the Evolution Control System is to provide automated support for coordinating the concurrent efforts of a team of prototype designers, and to manage multiple versions of the designs they produce [Bad93]. The purpose of the Merger is to combine the effects of two or more enhancements to a prototype that have been independently developed [Dam94].

CAPS can be executed in either the designer mode or the manager mode. The manager mode provides access to CAPS advanced features, including modification of the designer pool, creation of project work steps, and prototype change-merging. CAPS supports distributed prototype development, and the manager interface provides facilities for such efforts. For simple, single-designer prototype building, the designer mode should be used.

1. CAPS Tools

This section provides a brief description of each CAPS tool.

a. The PSDL Editor

The PSDL Editor is the heart of CAPS prototype design. This editor consists of 3 separate parts: the Syntax Directed Editor, the Graph Viewer, and the Graphic Editor. This tool allows the designer to create the CAPS data flow diagram and the PSDL program, and assign all timing and control constraints to prototype components (operators and data streams).

b. The Text Editor

Although the text editor is not exclusively a CAPS tool, CAPS does provide fluid integration of text editing facilities. Designers can select from vi, emacs and

the Verdix Ada Syntax Directed Editor (if available) for editing Ada programs. Use the "CAPS Defaults" selection under the "CAPS Edit" pull-down menu to make this selection. The CAPS User-Interface provides convenient file selection lists, based on the currently selected prototype.

c.   The Interface Editor

CAPS integrates TAE+ [Tae93] for creation of window-based user-interfaces for prototypes. When using the TAE Workbench for creation of such user-interfaces, the designer must use the "single file" Ada code generation option from within TAE+. The automatically generated TAE code is placed in the prototype directory in a file called

<prototype_name>.RAW_TAE_INTERFACE.a.

For details about how to integrate this file into a prototype, see Chapter VII of the CAPS Tutorial by Brockett [Bro94].

d.   The Requirements Editor

The current version of CAPS does not have a sophisticated requirements tracking or editing tool. Simple text editor integration is provided for editing requirements documents associated with a prototype. CAPS will automatically present the user with a list of all files with a ".req" suffix when "Requirements" is selected from the "Edit" pull-down menu. After a file is selected, the default text editor will be invoked on that file.

e.   The Change Request Editor

As with requirements, the current version of CAPS does not have a sophisticated change request tracking or editing tool. Simple text editor integration is provided for editing change request documents associated with a prototype. CAPS will automatically present the user with a list of all files with a ".cr" suffix when "Change Request" is selected from the "Edit" pull-down menu. After a file is selected, the default text editor will be invoked on that file.

23

f. The Translator

The CAPS translator converts a PSDL program into compilable Ada packages which implement supervisory aspects of the prototype. The translator expects a complete PSDL program as input, and creates several packages which make up, in part, the supervisor module of the prototype. It is important to note that the translator does not create Ada implementation packages for atomic operators or user-defined data types. These must be either extracted from the software base, or custom-made by the designer.

g. The Scheduler

The scheduler determines schedule feasibility for CAPS prototypes. Information is provided to the scheduler via timing constraints from the prototype's PSDL program. A prototype must be translated before it can be scheduled, and scheduled before it can be compiled. Upon scheduling a prototype, CAPS provides schedule diagnostic information which can be analyzed and used to direct timing constraint modifications.

h. The Compiler

CAPS uses the SunAda Ada compiler. The compilation process is completely automated via the "Compile" command provided in the "Exec Support" pull-down menu in the CAPS User-Interface. Successful prototype compilation requires the formal parameter lists of atomic operator implementation modules to conform to CAPS interface conventions.

i. The Evolution Control System

The CAPS Evolution Control System (ECS) [Bad93] is a system that supports distributed prototype development in a team environment. The ECS makes use of a design database (DDB) for persistent storage of prototype development data. The ECS supports maintenance of a designer pool from which to draw for prototype development tasks. Within the ECS, prototype development is modeled as a series of

steps that are created by the project manager. These steps are automatically scheduled and assigned to available designers.

j.   *The Merger*

The CAPS Merger [Dam94] provides automated prototype change-merging. Based on slicing theory, as applied to PSDL programs, the Merger automates the combination of two separate modifications to a base prototype. The Merger detects and warns of conflicts between the two changes to be merged. If no conflicts occur, or if they are overridden, the Merger creates a PSDL program for the newly created prototype which incorporates the changes of each of the modified prototypes.

k.   *The Software Base*

The CAPS software base and its associated retrieval mechanism [Dol93] provide access to a repository of reusable Ada and PSDL components. The software base allows a designer to browse as well as query its components. Queries to the software base can be in the form of keywords or PSDL specifications. In the current release of CAPS, the software base matching mechanism is based on parameter matching.

E.   THE PROTOTYPING SYSTEM DESIGN LANGUAGE (PSDL)

PSDL is a partially graphical specification language developed for designing real-time systems. It has several facilities for modeling timing and control constraints, but is also useful for requirements analysis and feasibility studies. It was designed as a prototyping language specifically for CAPS, to provide the designer with a simple way to specify software systems [LBY88]. PSDL places strong emphasis on modularity, simplicity, reuse, adaptability, abstraction, and requirements tracing.

A PSDL prototype is built as an hierarchical structure of components, graphically represented as data flow diagrams, and augmented with timing and control information. Each component may contain zero or more definitions for OPERATORS and TYPES, where each definition has two parts:

- *Specification part*: Defines the external interfaces of the operator or the type through a series of interface declarations, provides timing constraints, and describes functionality by using informal descriptions and axioms.

- *Implementation part*: Denotes what the implementation of the component is going to be, either in Ada or PSDL. Ada implementations point to Ada modules, which provide the functionality required by the component's specification. PSDL implementations are data flow diagrams augmented with a set of data stream definitions and a set of control constraints.

1. PSDL Computational Model

PSDL is based on a computational model containing OPERATORS that communicate via DATA STREAMS, where each stream carries values of a fixed abstract data type. There are several ADTs already built into PSDL; the PSDL_EXCEPTION is one of them. Modularity is supported through the use of independent operators that can only gain access to other operators when they are connected via data streams.

The PSDL computational model is formally represented as an augmented graph [LBY88]

$$G = (V,E,T(v),C(v))$$

where:

- V is a set of *vertices*
- E is a set of *edges*
- $T(v)$ is the set of *timing constraints* for each vertex v
- $C(v)$ is the set of *control constraints* for each vertex v

Each vertex represents an operator and each edge represents a data stream.

a. Operators

An operator represents either a function or a state machine. When it fires, an operator reads one data object from *each* of its input data streams and writes *at most one* data object on each of its output streams. If the output depends only on the current

26

set of input values, then the operator represents a function. In other words, the same response is given each time they are triggered. If, on the other hand, the output of the operator depends upon the input values and on internal state values representing some part of the history of the computation, then the operator represents a state machine.

A PSDL operator can be either atomic or composite. Operators that are decomposed into lower levels are called composite operators, and they represent networks of components. This decomposition is always functional. An operator that is not decomposed is called atomic, and in the current version of CAPS, they are implemented in Ada, but any language could be used for that purpose. According to the PSDL grammar, it is in the implementation part of the operator that we can declare an operator to be atomic or composite.

b. Data Streams

Data streams represent sequential data flow mechanisms which move data between operators. There are two kinds of data streams: sampled streams and data flow streams.

In PSDL the data trigger of a consumer operator determines the type of a data stream. If the stream is declared in the "TRIGGERED BY ALL" clause of the consumer operator, then the stream is a data flow stream. In all other cases it is a sampled stream.

Data-flow streams in the current implementation are similar to FIFO queues with a length of one. Any value placed into the queue must be read by another operator before any other data value may be placed into the queue, or it will overflow. Values read from the queue are removed from the queue, and if any attempt is made to read from an empty queue, it will underflow. Sampled data streams may be considered as a programming variable which may be written to or read from at any time and as often as desired. A value is on the stream until it is replaced by another value. Some values may never be read, because they are replaced before the stream is sampled. As can be seen, care must be taken when reading values from uninitialized sampled streams. All PSDL

27

data streams contain, at most, one data item at any given time. In summary, it could be said that a data flow stream guarantees that none of the data values are lost or replicated, while a sampled stream does not make such a guarantee.

c. State Streams

A CAPS prototype is a well-formed PSDL program if its graph representation (excluding all state streams) is a directed acyclic graph (DAG). This restriction may not seem to make sense at first glance. However, when a prototype graph contains a cycle, this indicates the presence of state information, and states must be explicitly declared and initialized. PSDL fully supports the integration of states in its prototypes.

When a state is introduced into an atomic operator, it must be implemented within the Ada code for that operator, and shouldn't appear in the graph as a self loop state edge.

d. Types

PSDL user-defined data types are abstract data types (ADTs) which can be used in CAPS prototypes. PSDL types, like PSDL operators, can be implemented in either PSDL or Ada. Types can be associated with a set of operators. Types implemented in Ada are realized by an Ada package that defines a private type and a subprogram for each operator on that type.

e. Exceptions

Exceptions in PSDL are values that can be transmitted on data streams of the type "PSDL_EXCEPTION". During prototype execution, undeclared exceptions are transformed into PSDL exceptions of the type PSDL_EXCEPTION, which is a subtype of UNDECLARED_ADA_EXCEPTION. Exceptions can also be raised by explicitly declaring them in the control constraints part of the PSDL program for the prototype.

f. Timers

PSDL timers are software stopwatches that are used to record the length of time between events, or to control the duration the system spends in some particular state. They are declared in the implementation part of a root operator, and are governed by the control constraints "START TIMER", "STOP TIMER" and "RESET TIMER".

2. Control Abstractions

As a major property of real-time systems, periodic execution, as well as other timing related attributes, is supported explicitly. The order of execution is only partially specified, and is determined from the data flow relations given in the enhanced data flow diagrams, but also affected by the types of data triggers among operators.

There are several control aspects to be specified, such as whether the operator is periodic or sporadic, the triggering conditions, and the output guards.

a. Periodic and Sporadic Operators

PSDL supports both periodic and sporadic operators. Periodic operators are triggered by the scheduler at approximately regular time intervals, so that they start execution somewhere after the beginning of the period, and complete by some deadline, which defaults to the end of the period. Sporadic operators are triggered by the arrival of new data, and possibly at irregular time intervals.

b. Data Triggers

Any PSDL operator can have a data trigger, of which there are two kinds, as illustrated by the following examples:

OPERATOR P TRIGGERED BY ALL X, Y, Z

OPERATOR Q TRIGGERED BY SOME A, B

In the first example, the operator P is ready to fire whenever new data values have arrived on all three streams X, Y and Z (triggering set), although there may be other streams coming into the operator P, in which case the data values do not need to be

29

new. This means that the data streams associated with X, Y and Z are data flow streams. This kind of trigger should be used when the items in a stream represent discrete events (e.g., transactions on a bank account) rather than samples from a continuous source of data (e.g., a temperature sensor). This kind of trigger also ensures that the output of the operator is always based on fresh data for all of the inputs in the triggering set.

The most important design consideration when "BY ALL" triggers are used is management of the firing frequencies of the producing and consuming operators. The period of the consuming operator must be smaller or equal to the period of the producing operator, or stream buffer overflow errors will result (i.e., the consuming operator must fire at least as often as the producing operator). This is because the data streams in CAPS can hold a maximum of one data item. CAPS ensures that if the consuming operator's period is less than that of the producing operator, the actual firing rate of the two will be the same (i.e., "BY ALL" trigger data streams are tested for new information prior to the actual firing of the consuming operator).

In the second example, the operator Q is ready to fire whenever new data arrives on at least one of the inputs A or B. This kind of activation condition guarantees that the output of operator Q is based on the most recent data from at least one of its critical inputs A and B, mentioned after the TRIGGERED BY SOME clause. This is also a very constrained condition, since the scheduler must guarantee that a new data in A or B will not be lost.

If a periodic operator has a data trigger, the operator is conditionally executed with the data trigger serving as input guard.

If a data trigger is not satisfied, the values are not read and, consequently, not consumed from any of the input streams.

c. Execution Guards

The firing of a PSDL operator can be regulated by an execution guard. Execution guards are conditional statements which are evaluated prior to firing the associated operator. Execution guards can depend on data from any incoming data stream

and they can be combined with the "BY ALL" and "BY SOME" data triggers mentioned above. Even if an execution guard is not satisfied, the values are read and consumed from all the input streams, without firing the operator. Examples are:

OPERATOR R TRIGGERED BY SOME X, Y IF X > 20.0

OPERATOR S TRIGGERED IF X: EXCEPTION

d. *Conditional Output*

PSDL conditional output is implemented in CAPS as guarded execution of code that writes values to data streams. Conditional output does not affect the firing of an operator, which will fire in accordance with the CAPS schedule regardless of whether or not its output is written to an output data stream. The condition of an output guard may depend on the output values of the operator, on the values read from the input streams, and on the values of timers.

3. Timing Constraints

Operators can be time-critical or non time-critical, depending on whether or not they are assigned a value for the maximum execution time (MET) by the designer. If time-critical, they can be further subdivided into periodic or sporadic operators. Periodic operators are explicitly assigned a frequency (PERIOD) of execution, meaning that they will fire within regular periods, exactly once, but not necessarily at regular intervals of time. Sporadic operators are not explicitly assigned a period, but they fire whenever there is new data on a set of input data streams, having, however, a minimum interval of time between successive firings. Periodic operators can also be triggered by the arrival of data. However, this trigger will behave like a condition to be checked during periodic firing. Every sporadic operator has an MRT and MCP in addition to an MET.

Timing constraints are an essential part of specifying real-time systems, and in PSDL the following timing constraints are supported:

- Maximum Execution Time (MET)
- Period (PER)

31

- Finish Within (FW)
- Maximum Response Time (MRT)
- Minimum Calling Period (MCP)
- Latency (LAT)
- Minimum Output Period (MOP)

The MET reflects the amount of CPU time that an operator may use for execution, and is applicable to both periodic and sporadic operators. Note that for atomic operators the MET complies with the above definition. For the composite operator, however, the MET is the maximum CPU time needed along any thread of control. Within CAPS, the MET is assumed to account for the following: data triggering checks, stream reads, execution guards checks, the execution itself, output guards checks, stream writes, and exception handling.

This parameter is by itself one of the most difficult to quantify. It is, therefore, unfortunate that it is also one of the most important parameters employed during the scheduling process. Two alternatives can be taken: to use the worst-case execution times, which can result in a poor processor utilization, or to use some value smaller than the worst-case, which introduces the possibility of an overload. For reasons of safety, CAPS uses the first approach by defining the MET as an upper-bound on the execution time. For further reading about execution time issues refer to Leinbaugh [Lei80, LY82] and Mok [Mok83].

Actually, due to the critical nature of the systems that CAPS was intended to prototype, the worst-case approach has been used throughout its design. This approach is observable even in the scheduling model, where the non-preemption option was chosen. This is because, while it is true that if a non-preemptive schedule can be devised for a set of tasks, then, it is possible to devise a preemptive one, but the opposite is not always true [Bla76].

The MRT defines an upper-bound on the time between the arrival of new data that satisfies all data triggering conditions of a sporadic operator and the time when the last value is written onto the output stream. The MRT applies only to sporadic operators.

The MCP also applies only to sporadic operators, and represents a lower-bound on the time between two consecutive triggerings of a sporadic operator. It constrains the behavior of the producers of the triggering data values, rather than constraining the behavior of the operator itself. Both timing constraints are illustrated in Figure 2.5.

As shall be seen later, each sporadic operator is going to be converted into an equivalent periodic one, whose period is called the triggering period (TP).

Scheduling delay for a sporadic operator is the interval of time between the writing into an output data stream by the producer and the corresponding reading of the input values by the consumer.
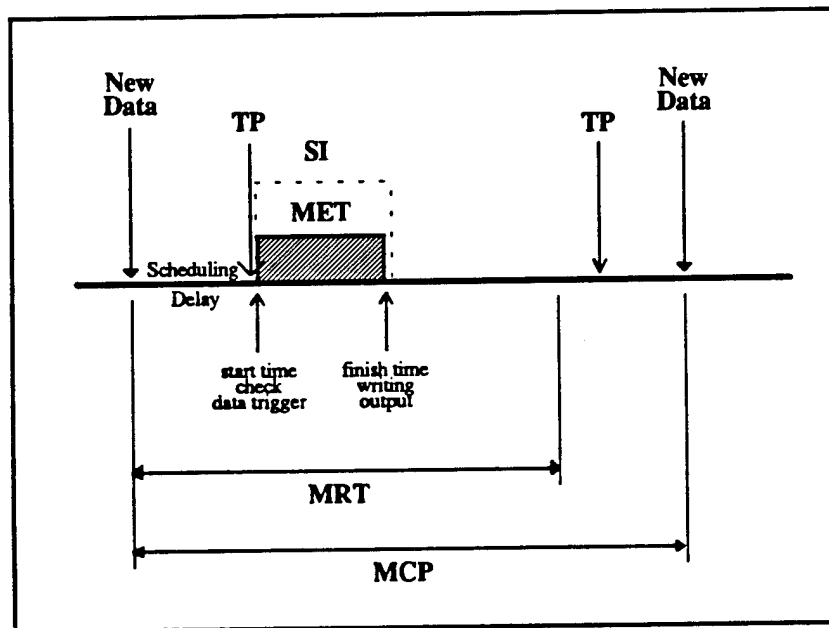
Figure 2.5. Sporadic Timing Constraints

Periodic operators are triggered by temporal events which must occur at regular intervals. For each operator, these activation times are determined by the specified period (PER), which is the time interval between two successive activations. The period applies

33

only to periodic operators. Note, however, that there is a distinction between activation time and the actual start time of a periodic operator as shown in Figure 2.6.
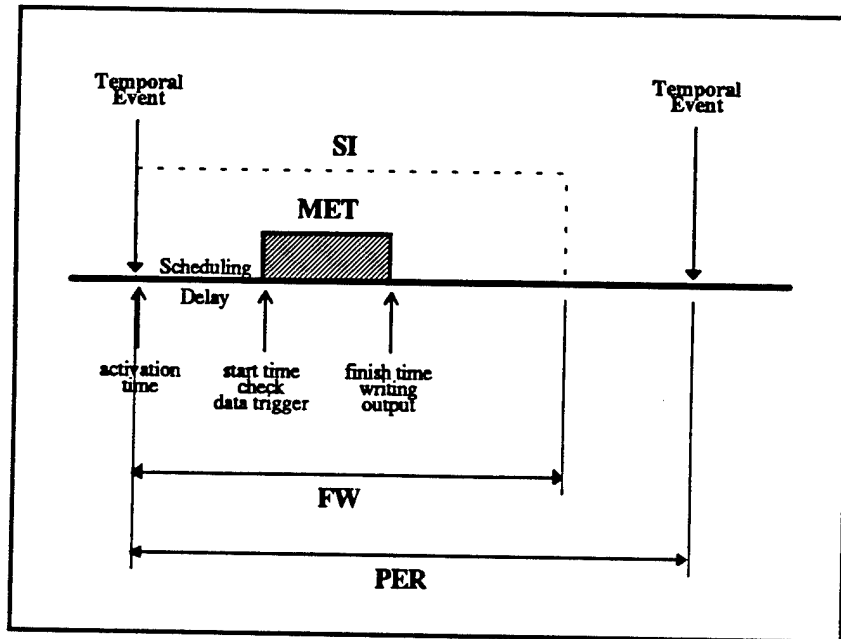
Figure 2.6. Periodic Timing Constraints

Finish within (FW) defines an upper bound on the finish time for a periodic operator. The difference between the activation time and its deadline is called the scheduling interval (SI) and it is equal to FW.

Scheduling intervals of a periodic operator can be viewed as fixed windows of a size equal to FW, evenly separated by the period PER, and whose absolute position on the time axis is determined by the start time t of its first execution. For the first instance this time may vary within the closed interval [0,PER] of the operator, and is called the *phase* of the operator (Figure 2.7). Scheduling intervals for sporadic operators will be covered in the next chapter, after we discuss how to deal with this type of operator.
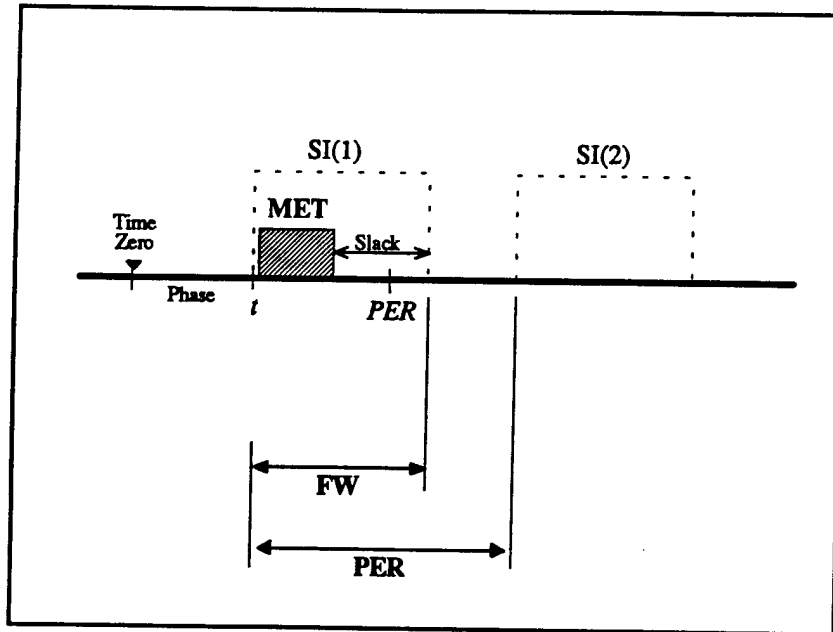
Figure 2.7. The Scheduling Interval

The difference between FW and MET is called the *slack* of the operator. Table 2.1 summarizes the timing constraints for periodic and sporadic operators.

Periodic Operators	Sporadic Operators
Maximum Execution Time (MET)	Maximum Execution Time (MET)
Period (PER)	Minimum Calling Period (MCP)
Finish Within (FW)	Maximum Response Time (MRT)

Table 2.1. Main PSDL Timing Constraints

To express the behavior of distributed systems, PSDL provides two timing constraints, Latency (LAT) and the Minimum Output Period (MOP). The latency of a stream is an upper-bound on the duration of the time interval between the instant a data value is written into a stream and the instant that data value becomes available for reading from the stream. In other words, the latency attribute for a stream is meant to specify an upper-bound on the allowable time spent by that stream in the network. This information should be used by the scheduler to simulate the worst case behavior for the delay in the network. Note, however, that this attribute does not explicitly require that the data

35

carried by the stream should be consumed, within the time interval, by the consumer operator on the other side of the network. The notation LAT_{xy} will be used to denote the latency associated with the stream between operators T_x and T_y.

The minimum output period is a lower-bound on the duration of the interval between two successive write events on the stream. In the absence of explicit synchronization, both the latency and minimum output period of a stream have the default value of zero (no delay, unbounded data rate). The purpose of these additional constraints is to declare communication constraints that arise from hardware limitations imposed by external constraints on how the software functions must be allocated to different physical nodes of a distributed system. Explicit modeling of these constraints is also sometimes required to ensure feasibility, because latency affects calculations of time budgets, as well as maximum execution times for composite operators. The effect of these constraints on static scheduling is that data cannot be read from a stream until a delay equal to the latency has elapsed, and that data cannot be written into a stream until the minimum period has elapsed.

4. A PSDL Prototype Example

Figure 2.8 shows a simple autopilot system that illustrates some of the typical features of PSDL. The example has a minimal specification part with an informal description. The implementation part contains a graph, making the operator Autopilot a "composite" operator. The figure also indicates maximum execution times, 170 ms for operator display, 50 ms for operators compass and altimeter, and 75 ms for the remaining operators. All operators are periodic with a period of 500 ms, except for the operator control_surfaces, which is sporadic, with an MRT and MCP of 900 ms, as it is shown in the control constraints part of the PSDL program.

Concluding, it can be said that the operator control_surfaces will be triggered whenever there is new data in either the course_command or the altitude_command streams. The operators correct_altitude and correct_course will be triggered whenever there is new data in the actual_altitude and actual_course streams, respectively.

36

OPERATOR autopilot
    SPECIFICATION
        STATES delta_course: INTEGER INITIALLY 0
        STATES delta_altitude: INTEGER INITIALLY 0
        STATES desired_course: INTEGER INITIALLY 0
        STATES desired_altitude: INTEGER INITIALLY 0
END

IMPLEMENTATION
    GRAPH

DATA STREAM
    actual_altitude: INTEGER,
    actual_course: INTEGER,
    altitude_command: altitude_command_type,
    course_command: course_command_type,
    elevator_status: elevator_status_type,
    rudder_status: rudder_status_type
CONTROL CONSTRAINTS
    OPERATOR altimeter
        PERIOD 500 MS
    OPERATOR compass
        PERIOD 500 MS
    OPERATOR control_surfaces TRIGGERED BY SOME course_command, altitude_command
        MAXIMUM RESPONSE TIME 900 MS
        MINIMUM CALLING PERIOD 900 MS
    OPERATOR correct_altitude TRIGGERED BY ALL actual_altitude
        PERIOD 500 MS
    OPERATOR correct_course TRIGGERED BY ALL actual_course
        PERIOD 500 MS
    OPERATOR display
        PERIOD 500 MS
END

Figure 2.8. Prototype of an Autopilot

37

III. FUNDAMENTAL ISSUES IN REAL-TIME SCHEDULING

A. THE SCHEDULING MODEL AND SOME DEFINITIONS

An instance of a prototype T can be thought of as the union of three disjoint finite sets, namely the set P of periodic operators, the set S of sporadic operators and the set N of non-time critical operators. Within CAPS, each periodic operator can be described, for scheduling purposes, as a three-tuple (MET_x, PER_x, FW_x), where MET_x is the maximum execution time used by each instance of operator X, PER_x is its period and FW_x is the length of its scheduling interval. Likewise, each sporadic operator can be described as a three-tuple $(MET_x, MCP_x, MRT_x)^{SP}$, where MCP_x is the minimum period between two consecutive instances of operator X, and MRT_x is the upper bound on the time between the triggering of operator X by some new data arrival, and the completion of writing to all of its output streams. The superscript SP is used in the sporadic case, only to distinguish from the three-tuple of the periodic operator. Given any static schedule for a prototype T, we shall use s_{ix}, f_{ix} and d_{ix} to denote the actual starting time, completion time and deadline of the ith instance of operator X in the schedule. In any feasible schedule, we must have

$$0 \leq s_{1x} \leq PER_x$$

and

$$d_{ix} = s_{1x} + (i - 1) \times PER_x + FW_x \qquad \text{Eq. (1)}$$

for every periodic operator X, where s_{1x} is called the phase of operator X as defined in Chapter II. Note also from Eq. 1 that the deadline for the *first instance* of any operator is calculated relative to its start time rather than from time zero[1]. This condition will release the scheduler from enforcing the condition that the first instance of operator X should finish by the time PER_x. Whenever possible, it is going to be used the letters X and Y to denote operators, leaving the letters i and j to denote their corresponding instances.

---

[1]Time zero is defined as the time when prototype starts execution. In reality it is the start time of the first operator according to the topological sort.

Since, in general, the release time does not affect the complexity of the scheduling problem [Mok83], it will be assumed that all first instances are released at time zero, but may be constrained by the precedence relationship between the operators, if one exists.

By definition, every periodic operator must start and finish execution within its period of activation.

The following restriction is also imposed on the model, where the maximum execution time must be smaller or equal to the finish-within, which in turn must be smaller or equal to the period:

$$MET \leq FW < PER$$

Clearly, the first inequality is needed, otherwise there is no way to execute such an operator within the specified amount of time (FW).

One may want to argue that there is a need to relax the second inequality to $PER <$ $MET \leq FW$. Since $PER < MET$, such processor demand can only be satisfied using pipelining in a multiprocessor environment [Luq93, LSB93], which will be discussed in the next section.

Note that for the sporadic operator all of the above assumptions are also applicable, since they will be converted into equivalent periodic operators, as can be seen later in this chapter.

The Harmonic Block (HB) of a periodic task set P is the least common multiple (LCM) of all the periods in P. It is the interval upon which the task set will be tested for schedulability. If a feasible schedule can be found within 2×HB, in the case where latencies are not allowed in the schedule, or in at most 3×LCM if latencies are allowed, then it is possible to say that the same pattern can be repeated forever. This topic will be further discussed in Section C.

A prototype T is said to be schedulable if there exists a schedule such that the completion time for the execution of instance i of operator X (f_{ix}) is less than or equal to its corresponding deadline d_{ix}, for all i and X, and the precedence constraints of the prototype T are satisfied.

The precedence constraint between operators X and Y, written as X < Y, where <
denotes a partial ordering on the execution of tasks X and Y, is satisfied if

$$\forall \text{ instances } i,j \qquad (i\text{-}1) \times PER_x + s_{1x} < (j\text{-}1) \times PER_y + s_{1y}$$

and

$$(j\text{-}1) \times PER_y + s_{1y} + \Delta < i \times PER_x + s_{1x}$$

where $(i\text{-}1) \times PER_x = (j\text{-}1) \times PER_y$ [2] and Δ equal the maximum time to read input by
operator Y.

Operators from either the periodic set P or from the sporadic set S are non-
preemptable, which means that once they start execution they will run to completion. The
only operators that can be preempted are those belonging to the set N.

No idle time is inserted into the static schedule, unless there are no operators ready
to execute.

All timing information is assumed to be an integral multiple of a basic unit of time,
which within CAPS is assumed to be the millisecond. Table 3.1 presents a summary of the
major assumptions of the scheduling model.

For all periodic operators $MET \leq FW \leq PER$
All time-critical operators are non-preemptable
Time is discrete
A periodic operator is completely specified by the tuple (MET, PER, FW)
A sporadic operator is completely specified by the tuple (MET, MCP, MRT)^{SP}
Static Scheduling is assumed

Table 3.1. Summary of our Scheduling Model

In the next section, a series of theorems on schedulability for a set of independent
non-preemptive periodic task sets will be presented. They will provide the necessary
background to build a framework upon which the later sections of this chapter will be
based.

---

[2] This condition will be relaxed after we present our new synchronization model in Chapter IV.

B. CONDITIONS FOR SCHEDULABILITY OF NON-PREEMPTIVE TASKS

In this section, a series of schedulability checks are introduced for a periodic task set P that has no precedence constraints. These results will be also applied to a set of periodic tasks with precedence constraints in Section D of this chapter.

1. The Maximum Execution Time Theorem

When dealing with non-preemptive *uniprocessor* static scheduling a sufficient condition for unfeasibility occurs whenever a task requires more computation time than the period of any other task, or more specifically, more than the minimum period among all tasks. Formally:

Theorem 1:

"For an independent periodic task set P, if $\exists$ some tasks X and Y $\in$ P, such that $MET_x \geq PER_y$, then P is not schedulable in the uniprocessor case by any non-preemptive algorithm. Furthermore, if X = Y then neither the preemptive nor the non-preemptive algorithms can find a feasible schedule."

Proof:

Clearly, whenever task X executes, task Y, which happens to have a smaller period, will be blocked for an interval of time bigger than its period, which is contradictory with the definition of a periodic task.   ❑

Note that the Theorem still holds if precedence relationship exists among the tasks in P. This same result is also valid for a sporadic task set when $MET_x > MCP_y$ for X = Y (trivial case). However, for X ≠ Y the situation is slightly more complex, and there are two cases to consider. The first is when $MRT_y < MCP_y$, and it is clearly not schedulable. The second case is when $MRT_y \geq MCP_y$, and the set is not schedulable if $MET_x + MET_y > MRT_y$, as shown in Figure 3.1.
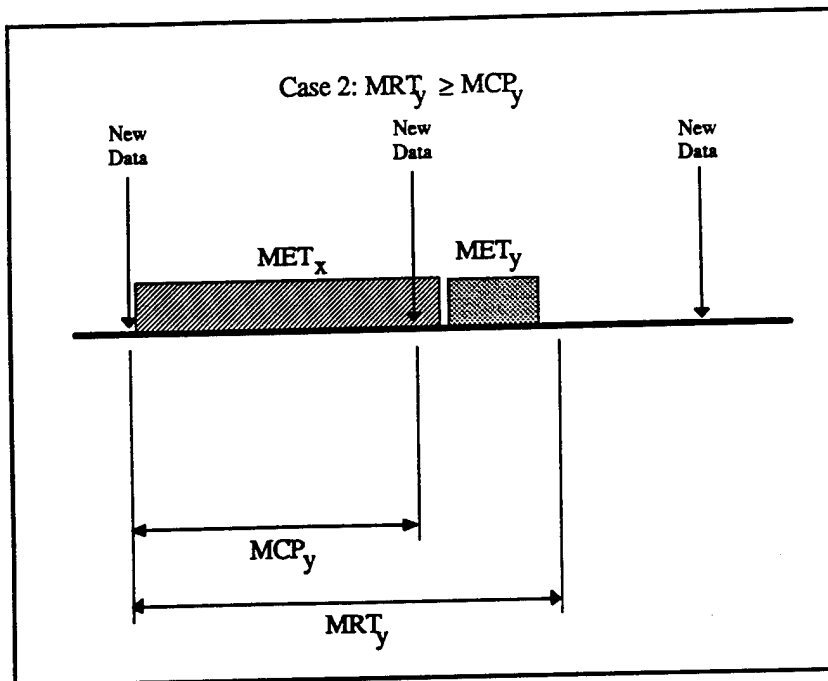
Figure 3.1. Theorem 1 for the Sporadic Case

Corollary: (for the distributed case)

"For an independent periodic task set P, if $\exists$ some tasks X and Y $\in$ P, such that $MET_x \geq PER_y$, then in order for P to be schedulable in the multiprocessor case, tasks X and Y must be placed in different processors, and if X = Y, then it must be pipelined." $\square$

The conditions imposed on a task X for it to be pipelineable as well as a detailed description of pipelining in this context, can be found in the work of Luqi [Luq93] and Luqi, Shing and Brockett [LSB93].

There are two ways to handle pipelining. The first is to use task migration at run-time, which involves sending a copy of the code and data to be executed in the other processor. This presents the following problems:

1) It increases the context switching overhead, with direct impact on the timing constraints

2) There is a need to create an additional task to handle the dispatching of tasks

3) It is not well suited for static scheduling

The second approach is to replace the tasks to be pipelined in the other processors in a pre-processing step. For example, consider a periodic operator $OP_A(150,100,150)$ with inputs D1, D2 and output D3 as shown in Figure 3.2. As shown in Figure 3.2b, we can replace operator OP_A with two identical operators, $OP_B(150,200,150)$ and $OP_C(150,200,150)$, with twice the original period and a state stream *syn*, whose latency equals the time taken by the non-overlappable segment of the code implementing operator OP_A. The operators OP_B and OP_C will be triggered alternately on the value of *syn*.
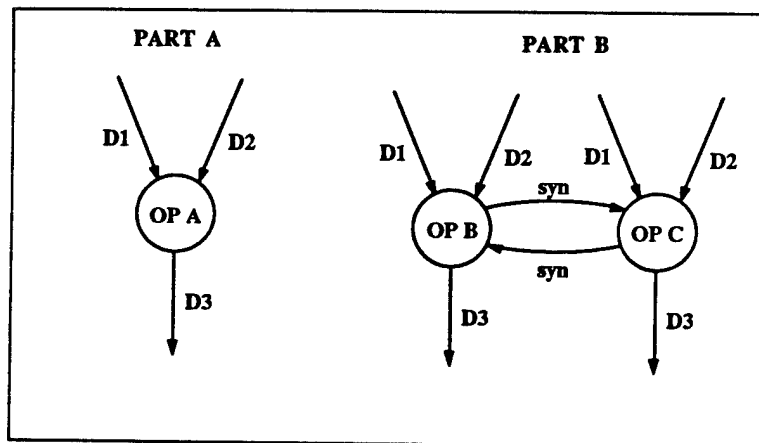
Figure 3.2. Pipelining Operators

The replication of tasks throughout the system presents the following problems:

1) It increases the memory requirements for the processors

2) It demands highly sophisticated mechanisms for implementing tight synchronized schedules among the processors, which restricts this approach to the shared memory models with a global clock

Both of the above discussed methods, however, suffer from the very serious problem of having to quantify the timing parameters of the segments of code that cannot be overlapped, which is by itself one of the hardest ones. If those timing parameters could be known in advance, then the operator could be separated into independent parts, and pipelining would not be needed.

The validity of pipelining in a hard real-time environment is therefore questionable, and, furthermore, it is impossible to implement in a distributed system where there is no inexpensive method by which to assure tight synchronization among tasks.

2. The Finish-Within Theorem

Theorem 2:

"For an independent periodic task set P if $\exists$ some indivisible task $X \in P$ such that $MET_x > FW_x$ then P is not schedulable under any scheduling algorithm, not even in a multiprocessor environment."

Proof:

Clearly, if $MET_x > FW_x$, the only way to handle this case is if we could split task X into two or more data independent partitions, so that they could run in parallel on different processors, but, as stated in the theorem, X is indivisible. $\square$

Note that this theorem can be easily extended to cover the sporadic case when $MET_x > MRT_x$. It is also applicable to the case where we have precedence constraints in the set P.

3. The Minimum Period Theorems

In the other extreme of Theorem 1, there is a sufficient but not necessary condition to guarantee schedulability of an independent periodic task set, as stated in Theorem 3:

Theorem 3:

"For a periodic task set P, if $\forall$ tasks $X \in P$, $FW_x \geq PER_x$ and $\sum_{x=1}^{n} MET_x \leq PER_z$

where PER_z denotes the minimum period in P, then P is schedulable." [3]

Proof:

The minimum period is certainly a divisor of the least common multiple of the periods (LCM), and, as such, it can span the entire LCM within an integral number of

---

[3]Similar result was achieved independently by Zhu, et al. [ZLC94] using the concept of critical time section.

45

steps. It is a kind of sliding bin-packing where a sliding window of size equal to the minimum period is present and, always large enough to fit all tasks present in that window. Of course, depending on the periods, all instances may not be active simultaneously in that specific window. However, in the event that it does happen, the instances will always fit in there. ❏

As shall be seen later, this theorem is valid even when precedence constraints are taken into consideration.
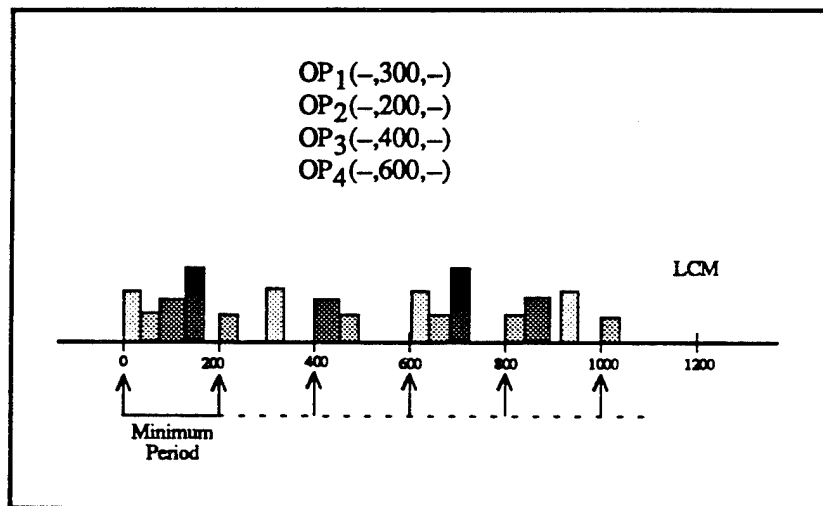
Figure 3.3. The Minimum Period Sliding Window

It is possible to use a counter example to show that the above condition is a sufficient but not necessary condition. Consider two periodic tasks with the following timing constraints: (5,10,10) and (2.5,5,5). The sum of METs is bigger than the minimum period, but this task set is still schedulable.

What happens if all deadlines are restricted to be less than or equal to their corresponding periods? In this case it could be said that Theorem 3 is not applicable, as illustrated by the following example: (3,5,3), (1,10,3).

Theorem 4:

"For a periodic task set P, if $\forall$ tasks $X \in P$, $\sum\limits_{x=1}^{n} MET_x \leq FW_z$, where FW_z denotes the minimum FW in P, then P is schedulable."

Proof:

The same idea of sliding bin-packing applies here. Now, however, the size of the bin must be decreased. In other words, the "bin" now should be understood to be the least value among all periods and FW, among the tasks from P. $\square$

The next theorem to be presented is the Load Factor Theorem, which is very well known in the field of scheduling. It defines a necessary condition for the schedulability of a periodic task set, and it basically stipulates that if the summation of all individual load factors (MET_x/PER_x) is bigger than the number of available processors, then the set is not schedulable [LL73].

4.   The Load Factor Theorem

Theorem 5:

"For a periodic task set P, if $\sum\limits_{x=1}^{n} \dfrac{MET_x}{PER_x} > k$, where k is the number of available processors, then the set is not schedulable."

Proof:

A very simple proof is given independently by Zhu [ZLC94] and Jeffay [JSM91] for the case where k equals 1. Basically, if both sides of the inequality are multiplied by the least common multiple (LCM) of their periods, it does not affect the inequality, but now

$$\sum_{x=1}^{n} MET_x \times \frac{LCM}{PER_x} > LCM \qquad\qquad \text{Eq. (2)}$$

Clearly, the ratio LCM/PER_x defines an integer that represents the number of instances for each task X within the LCM. If the number of instances of each task is multiplied by its maximum execution time and the results are then added, the result is the

total computation time needed by the entire task set. According to Eq. 2, however, the total computation time needed is bigger than the LCM. In other words, even if all instances are executed one after another, they would not be able to finish within LCM. The case for k greater than one follows automatically.　　　　　□

It should also be clear from the proof of Theorem 5 that it is valid to both preemptive and non-preemptive algorithms [ZLC94].

5.　　The Task Demand Theorem

The following theorem is based upon the previous work of Jeffay, et al. [JSM91] which established necessary and sufficient conditions for schedulability of an independent periodic task set in a non-preemptable uniprocessor environment. The theorem to be introduced next is an adaptation for the scheduling model used in this dissertation. It differs from the original theorem in that Jeffay's model accounts for, tasks that are independent, there was no explicit deadline for the tasks other than their own period, and his definition for a schedulable set of tasks required that both conditions in the theorem should be valid for *every concrete task set* generated from P, where a concrete task set can be viewed as the original independent periodic task set P with specific release times for the first instance of every operator in P.

The inclusion of the deadline which differs from the corresponding period into the problem made it a lot more complex, since tasks can now finish as early as their MET. The new results are presented in the following theorems:

Theorem 6:

"For an independent periodic task set P, where the tasks are sorted in non-decreasing order by finish-within (i.e., for any pair of tasks X and Y, if X < Y, then $FW_x \leq FW_y$), if there exists a feasible schedule for every concrete task set in P, then the following conditions hold: "

1) $\sum\limits_{x=1}^{n} \dfrac{MET_x}{PER_x} \leq 1,$

48

2) $\forall x, 1 < x \leq n; \quad \forall k, 0 \leq k < \dfrac{LCM}{PER_x};$

$$\sum_{y=1}^{n} N(y, k \times PER_x + FW_x) \times MET_y \leq k \times PER_x + FW_x$$

3) $\forall x, 1 < x \leq n; \quad \forall L, FW_1 < L < FW_x;$

$$L \geq MET_x + \sum_{y=1}^{x-1} N(y, L\text{-}1) \times MET_y$$

where

$$N(y, L) = \begin{cases} \left\lfloor \dfrac{L}{PER_y} \right\rfloor & \text{if } L \bmod PER_y < FW_y \\ \left\lfloor \dfrac{L}{PER_y} \right\rfloor + 1 & \text{if } L \bmod PER_y \geq FW_y \end{cases}$$

and LCM is the least common multiple of all the periods of the periodic task set.

Proof:

Condition 1) is basically Theorem 5 for the uniprocessor case. Conditions 2) and 3) together say that for the set to be schedulable, the processor demand in the interval [0,L] (i.e., the sum of computation times from all instances that must finish in the interval [0,L]), must always be less than or equal to the length of L. As in Jeffay's work [JSM91], the contrapositive of Conditions 2) and 3) will be proven. To prove the contrapositive of Condition 2), consider a concrete set of periodic tasks $\{T_1, T_2, ..., T_n\}$ where for $1 \leq X \leq n$, the release time of the first instance of Task $T_x = 0$. Then, for every X, $1 \leq X \leq n$, and every k, $0 \leq k < \dfrac{LCM}{PER_x}$, the processor demand, $d_{0, k \times PER_x + FW_x}$, from all task instances that must finish in the interval $[0, k \times PER_x + FW_x]$ is given by

$$d_{0, k \times PER_x + FW_x} = \sum_{y=1}^{n} N(y, k \times PER_x + FW_x) \times MET_y$$

So if Condition 2) does not hold, then there exist an X and a k such that $d_{0, k \times PER_x + FW_x} > k \times PER_x + FW_x$ and P has an unschedulable concrete set.

To prove the contrapositive of Condition 3), consider a concrete set of periodic tasks $\{T_1, T_2, ..., T_n\}$ where for some task T_x, the release time of its first instance is $T_x =$

0, and for all $Y \neq X$, the release time of the first instance of task $T_y = 1$, as shown in Figure 3.4.
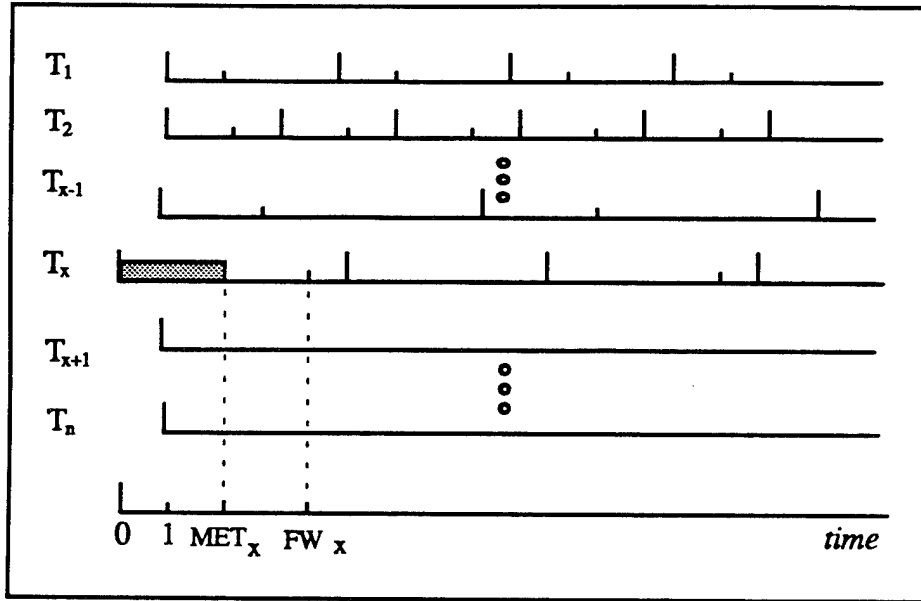
Figure 3.4. Different Task Release Time for Task X

Since neither preemption nor inserted idle time are allowed, the first instance of task T_x must execute in the interval $[0, MET_x]$. For all L, $FW_1 < L < FW_x$, in the interval $[0,L]$ the processor demand $d_{0,L}$, from all task instances that must finish by time L, is given by

$$d_{0,L} = MET_x + \sum_{y=1}^{x-1} N(y,L-1) \times MET_y$$

So, if Condition 3) does not hold, then $d_{0,L} > L$, and P has an unschedulable concrete set. $\square$

Note also that the function $N(y,L)$ can also be expressed in closed form as follows:

$$N(y,L) = \left\lfloor \frac{L}{PER_y} \right\rfloor + \min\left(\left\lfloor \frac{L}{FW_y + \left\lfloor \frac{L}{PER_y} \right\rfloor \times PER_y} \right\rfloor, 1 \right)$$

The left hand side of the addition operator specifies how many full periods there exist for task y within L, while the right hand side specifies whether the remaining fraction of a whole period is large enough for a scheduling interval (i.e., FWy) of task Y. The minimum comes into play because if $FW_y \leq L/2 < PER_y$, it would contribute more than once for the processor demand in the first period, which cannot occur.

As an example consider the task set $T_1(8,45,20)$, $T_2(9,40,30)$, and $T_3(10,100,100)$, already sorted by FW.

Clearly, n = 3 and the interval of interest is 20 < L < 100.

Let i = 1, then L = 20, which is the trivial case.

Let i = 2, then 20 < L < 30

for 20 < L < 30,	L must be ≥ 9 + 8	☑

Let i = 3, then 20 < L < 100

for 20 < L < 30,	L must be ≥ 10 + 8	☑
for 30 ≤ L < 65,	L must be ≥ 10 + 8 + 9	☑
for 65 ≤ L < 70,	L must be ≥ 10 + 8 + 8 + 9	☑
for 70 ≤ L < 100,	L must be ≥ 10 + 8 + 8 + 9 + 9	☑

If the task set was not approved in all conditions, it could be said that there exist at least one concrete task, that could not be scheduled. Alternatively, if all conditions were satisfied, then nothing else could be stated before Theorem 7 is introduced.

Theorem 7:

"If an independent periodic task set P is schedulable according to Theorem 6, then the non-preemptive Earliest Deadline First (EDF) algorithm will be able to find a feasible schedule for P."

Proof:

As in Jeffay's work [JSM91] this theorem shall be proved by contradiction. Assume that a task in P misses a deadline at some point in time when P is scheduled by the EDF algorithm. Let t_d be the earliest point in time at which a deadline is missed. All instances of P can be partitioned into three disjoint sets S_1, S_2 and S_3 where:

51

S_1 is the set of task instances with a deadline at t_d;

S_2 is the set of task instances with an invocation before t_d and deadlines after t_d, and

S_3 is the set of task instances not in S_1 or S_2.

Let t_0 be the end of the last period prior to t_d, in which the processor was idle. If the processor has never been idle, then $t_0 = 0$. Since neither preemption, nor inserted idle time are allowed, all task instances which are executed in the interval $[t_0, t_d]$ must be activated at or after t_0. Depending on whether the interval $[t_0, t_d]$ contains any task from the set S_2, the following two cases exist:

Case 1: None of the tasks in S_2 are scheduled in the interval $[t_0, t_d]$.

This case only happens if $t_0 = 0$. Otherwise, we either have an instance that misses its deadline in the interval $[0, t_0]$ if $t_0 - 0 > t_d - t_0$, or the processor has an idling period in the interval $[t_0, t_d]$, if $t_0 - 0 \leq t_d - t_0$. Furthermore, $t_d \leq LCM$. Otherwise, we must have another instance that misses its deadline prior to t_d.

Let T_{ix} be the task instance that misses the deadline at time t_d. Then, $t_d - 0 = k \times PER_x + FW_x$ for some k, $0 \leq k < \dfrac{LCM}{PER_x}$. The processor demand, $d_{0, k \times PER_x + FW_x}$, from all instances which must finish in the interval $[0, k \times PER_x + FW_x]$ equals

$$\sum_{y=1}^{n} N(y, k \times PER_x + FW_x) \times MET_y$$

and it is greater than $k \times PER_x + FW_x$, a contradiction.

Case 2: Some of the task instances of S_2 are scheduled to run in the interval $[t_0, t_d]$.

Let T_{ix} be the last instance in S_2 scheduled to run prior to t_d in the interval $[t_0, t_d]$ and let t_{ix} be the starting time of T_{ix}. The invocation time of all task instances scheduled to start in the interval $[t_{ix}+1, t_d]$ must be at or after $t_{ix}+1$ and with deadline at or before t_d, otherwise the EDF algorithm will not schedule T_{ix} to start at t_{ix}. Hence, the process demand for the interval $[t_{ix}, t_d]$, d_{t_{ix}, t_d}, must be bounded from above by the inequality

$$d_{t_{ix},t_d} \le MET_x + \sum_{y=1}^{x-1} N(y, t_d - (t_{ix}+1)) \times MET_y$$

Since there is no idle time in $[t_{ix}, t_d]$, and since a task missed a deadline at t_d, it follows that $d_{t_{ix},t_d} > t_d - t_{ix}$.

Let $L = t_d - t_{ix}$. Then

$$FW_1 < L < FW_x$$

and

$$L < d_{t_{ix},t_d} \le MET_x + \sum_{y=1}^{x-1} N(y, L-1) \times MET_y$$

contradicting condition 3 of Theorem 6. $\square$

Note that Condition 3 in Theorem 6 is a sufficient but not necessary condition for schedulability of a particular concrete task set, as illustrated by the following example. Consider the task set $T_1(100,150,150)$ and $T_2(100,300,200)$. Clearly it does not satisfy Condition 2, a feasible schedule may still be found if their release times are zero. However, if the release time of T_2 is changed by only one unit of time, then the set is no longer schedulable.

Jeffay, et al. [JSM91], have shown that the problem of determining whether a feasible schedule exists for a particular concrete task set is NP-Hard.

C.   THE HARMONIC BLOCK DILEMMA

It is a well known and accepted result that the least common multiple (LCM) of the periods of a periodic task set provides a finite interval of time, for which a cyclic schedule can be calculated, if one exists, and repeated forever [Mok83].

Many interpret the above statement to mean that a cyclic feasible schedule must only exist in the closed interval [0,LCM], i.e., a feasible schedule for all tasks instances that must start in the interval [0,LCM] and complete execution by time LCM. Such an interpretation holds only if the first instance of every task T_x is restricted to complete its execution by time PER_x. But what if such a restriction is not desirable? It seems very

53

reasonable to allow the first instance of a periodic task to start within its period of activation but finish up to the end of the period plus its computation time, and actually this would be a very desirable property, if it could somehow improve the already difficult problem of non-preemptive scheduling.

Consider the task set $T_1(190,600,600)$ and $T_2(20,200,200)$ with the precedence relation $T_1 < T_2$, as illustrated in Figure 3.5.
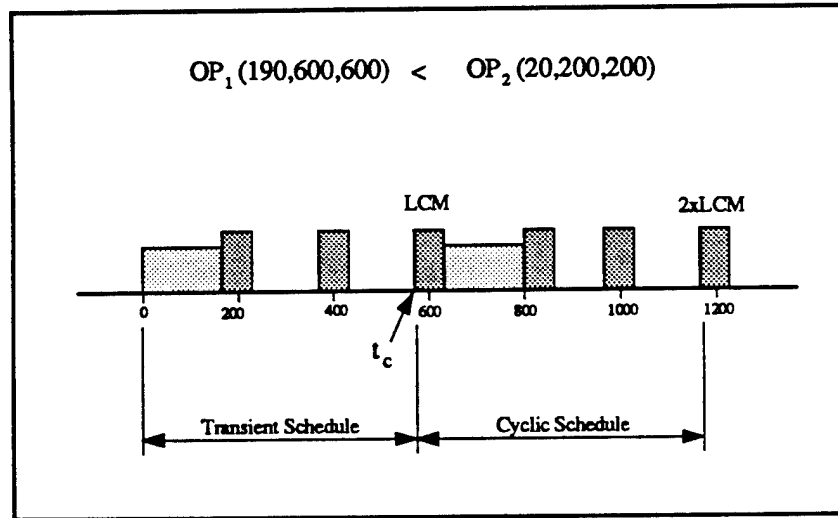
Figure 3.5. The Transient and Cyclic Schedules

Clearly, no feasible schedule exists if the first instance of every task T_x is restricted to complete its execution by time PER_x. However, if it is allowed to the first instance of every task T_x to start by time PER_x and complete its execution by time $PER_x + MET_x$, then a feasible schedule exists. Note also that the cyclic schedule no longer starts at time zero, but starts instead at time t_c, and furthermore, there can be more than one task instance that does not finish by time $2\times LCM$, as can be illustrated by the task set $T_1(4,100,100)$, $T_2(2,5,5)$, $T_3(2,100,100)$ and $T_4(3,10,10)$, with precedence relations $T_1 < T_2 < T_3 < T_4$.

Here is where a novel approach on how to determine what is a suitable cyclic schedule comes into play. The fundamental concept is that a feasible static schedule

54

consists of two parts: a transient part, which may be empty, followed by a cyclic part, which repeats forever.

The next theorem, the Harmonic Block Theorem, although different from the one introduced by Zhu, et al. [ZLC94], was created after a careful analysis of their work, which does not correctly solve the problem. The general direction of the proof will consist in showing that if the premises of Theorem 8 are satisfied, then there exists some time t_c where a part of the schedule can be divided, with exactly the size of one LCM, where it is guaranteed that the correct number of task instances are present, and most importantly, that they all start and finish within that time interval, characterizing the cyclic part of the new schedule.

Theorem 8: **The Harmonic Block Theorem**

"If $\exists$ an infinite feasible schedule S without any inserted idle time for a periodic task set P with precedence constraints, such that the first instance of every task, T_x in P must start by time PER_x, then there exists an infinite feasible schedule S' consisting of a transient portion of length at most LCM, followed by a cyclic portion of length LCM that repeats forever."

Proof:

If there is no idling time period in the intervals [0,LCM] or [LCM,2×LCM], then the given set of periodic tasks P must have a load factor of 1, and the first instance of every task T_x must finish its execution at or before time P_x in any feasible schedule. Hence, the segment of S in the interval [0,LCM] forms the cyclic portion of an infinite feasible schedule satisfying the Theorem.

Suppose now that idling time exists in the intervals [0,LCM] and [LCM,2×LCM]. Let t_c be the end of the last period prior to time LCM in which the processor was idling in S, and let t_1 be the end of the last period prior to time t_c+LCM in which the processor was also idling in S as shown in Figure 3.6.
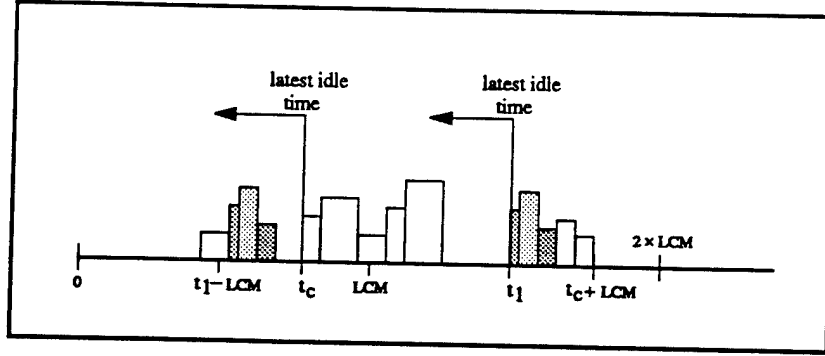
Figure 3.6. Determining the Start Time t_c of the Cyclic Schedule

Assertion (1)

Since no unnecessary idle time is inserted in our schedule S, it should be clear that there cannot be any *first instances* of tasks being activated after time t_c, because otherwise they could have started execution before time t_c.

Assertion (2)

Another important point to be made is that all tasks which start after time t_1 could not be activated before time t_1, for the same reasons of non-inserted idle time in our schedule S.

Assertion (3)

Every task instance that is activated in the interval $[t_1, t_c + LCM)$ must finish its execution at or before $t_c + LCM$. Suppose this claim is not true. Then there must exist some instances which are activated before $t_c + LCM$ and cannot finish at or before $t_c + LCM$. Denote the collection of all instances which are activated in the interval $[t_1, t_c + LCM)$ by τ. It follows from assertion (2) that every instance in τ must be activated in the interval $[t_1, t_c + LCM)$. This implies that

$$\sum_{T_{ix} \in \tau} MET_{ix} > t_c + LCM - t_1 \qquad (i)$$

Let τ' denote the set of task instances that are activated in the interval $[t_1 - LCM, t_c)$. It follows from assertion (1) that every task instance in τ must have a corresponding instance in τ'. Thus $|\tau| \leq |\tau'|$, and

$$\sum_{T_{ix} \in \tau} MET_{ix} \leq \sum_{T_{iy} \in \tau'} MET_{iy} \qquad (ii)$$

56

Note that all instances in τ' must finish within the interval $[t_1\text{-LCM}, t_c]$, because t_c is the end of an idling period. Hence,

$$\sum_{T_{iy}\in\tau'} \text{MET}_{iy} < t_c - (t_1\text{-LCM}) = t_c+\text{LCM}-t_1 \qquad\qquad \text{(iii)}$$

From inequalities (i), (ii), and (iii),

$$t_c+\text{LCM}-t_1 < \sum_{T_{ix}\in\tau} \text{MET}_{ix} < t_c+\text{LCM}-t_1 \ ,$$

which is a contradiction.

Assertion (4)

All instances after t_c are at least second instance and hence, for all tasks T_x within the interval $[t_c, t_c+\text{LCM})$, there must exist $\dfrac{\text{LCM}}{\text{PER}_x}$ activations. By assertion (3) they all finish within this same interval. The segment of S in the interval $[t_c, t_c+\text{LCM})$ contains the correct number of instances.

Concluding the proof, it can be said that the intervals $[0, t_c]$ and $[t_c, t_c+\text{LCM}]$ of S form respectively the transient portion and the cyclic portion of the new schedule S', satisfying the consequence of the Theorem. □

As can be seen, by a proper choice of the start time of the cyclic portion of the schedule, one can increase the schedulability of tasks sets which were previously assumed to have no feasible schedule, when the cyclic schedule was restricted to always start at time zero. Note also that the same approach is valid for preemptive task sets.

D.   A NOTE ABOUT PRECEDENCE CONSTRAINTS

Every reference to the word precedence constraints between tasks is usually attached to the meaning of synchronization, in other words, if two tasks have some kind of precedence relation, then they must be synchronized. Furthermore, if their periods are different, then they should be synchronized at intervals corresponding to the least common multiple of their periods. But then, what is the real need for synchronization if there are cases where some data may well be lost? Does it exist only to enforce a fixed pattern on how data are lost, e.g., instances three from task X and two from task Y, six and four and

57

so forth will synchronize? These and other questions will be much further discussed in Chapter IV.

We shall argue in Chapter IV that the major reason for synchronization is to guarantee timely processing of triggering data. We shall show that, by relaxing the upper bound on the delay in processing each instance of triggering data, we can guarantee that, even without explicit synchronization, each instance of the trigger data will be processed within an interval equal to two times the period of the consumer operator. The removal of the need for synchronization is particularly important in distributed systems, where synchronization mechanisms are very costly if not impossible. It is also desirable not to have synchronization in uni-processor systems, because now, we can treat each topological ordering of the tasks satisfying the precedence relationships as a concrete set of periodic tasks, where the starting time of task T_x is greater than or equal to the sum of the MET_y of all tasks T_y that are ancestors of T_x in the task graph.

Note that if non-zero latency is present in the edges of the precedence graph, then we must further delay the starting time of the first instances of every task Y, so that $S_{1y} \geq \max\{S_{1x}+MET_x+LAT_{xy}$, $\forall$ parent operator T_x of $T_y\}$, where LAT_{xy} denotes the latency associated with the edge (T_x , T_y).

In order for the arguments in the proof of Theorem 8 to hold, we need to choose t_c to be the end of the first idling period after time LCM, resulting in a Modified Harmonic Block Theorem that reads:

Theorem 9:

"If $\exists$ an infinite feasible schedule S for a periodic task set P with precedence constraints, such that the first instance of every task, T_y in P must start by time PER_y, then there exists an infinite feasible schedule S' consisting of a transient portion of length at most 2×LCM, followed by a cyclic portion of length LCM that repeats forever."

Proof:

The main difference when dealing with latencies, is that idling periods may exist before the starting time of the first instance of some task T_x in the schedule. Theorem 8

still holds for this case, because the presence of idling time only affects the release time of the tasks, as long as $PER_y \geq S_{1y} \geq \max\{ S_{1x}+MET_x+LAT_{xy} \}$. However, for Theorem 8 in Section C, the cyclic portion of the schedule may now start after time LCM. The reason is because the schedule S may contain first instances in the interval $[t_c, t_c+LCM]$, which was the key in our previous proof of Theorem 8. After these considerations, the same proof used for Theorem 8 can be applied to this case. □

E.   COPING WITH APERIODIC TASKS

Generally speaking, a sporadic task is defined as an aperiodic task that has a minimum duration between two consecutive activations. If that was not so, neither the static nor the dynamic approach could be used to guarantee schedulability.

If interrupts are used to detect the occurrence of aperiodic events at run-time, then a dynamic approach should be used. However, in the static scheduling framework, where all the tasks requests must be known a priori, so that a fixed and static schedule can be generated, the only way to handle sporadic tasks where we do not know exactly when they are going to happen, is by using a periodic process to function as a polling device. Its main role is to check for requests of sporadic tasks and to serve them during its allocated time slot. However, due to the random nature of aperiodic processes, we may not be able to handle a concentrated set of arrivals or even worse, not catch them at all with the sporadic server approach. To overcome this difficulty, several bandwidth preserving algorithms have been proposed. Among them could be mentioned the Priority Exchange, Deferrable Server and the Sporadic Server. [AB93]

The CAPS approach was to use one sporadic server for each time-critical sporadic operator. This approach, although very restrictive, is the only way to guarantee that all time-critical sporadic tasks would be serviced in a timely fashion under the worst case situation.

Therefore, the next step is to convert the sporadic operator into a periodic one so that all the original timing constraints from the sporadic operator are still satisfied.

1. The Conversion

The term triggering period (TP) will be used for the period of the converted sporadic operator and the usual term FW for its finish-within. As shown in Figures 3.7 and 3.8, basically two cases can occur:

The first is when MCP < MRT - MET and the equivalent periodic operator must have TP ≤ MCP in order to satisfy the original timing constraints. Also, must enforce that FW = MRT - MCP, so that in the critical case shown in Figure 3.7, the data that was missed by the previous triggering period can be consumed by the next TP and still finish within the original MRT.
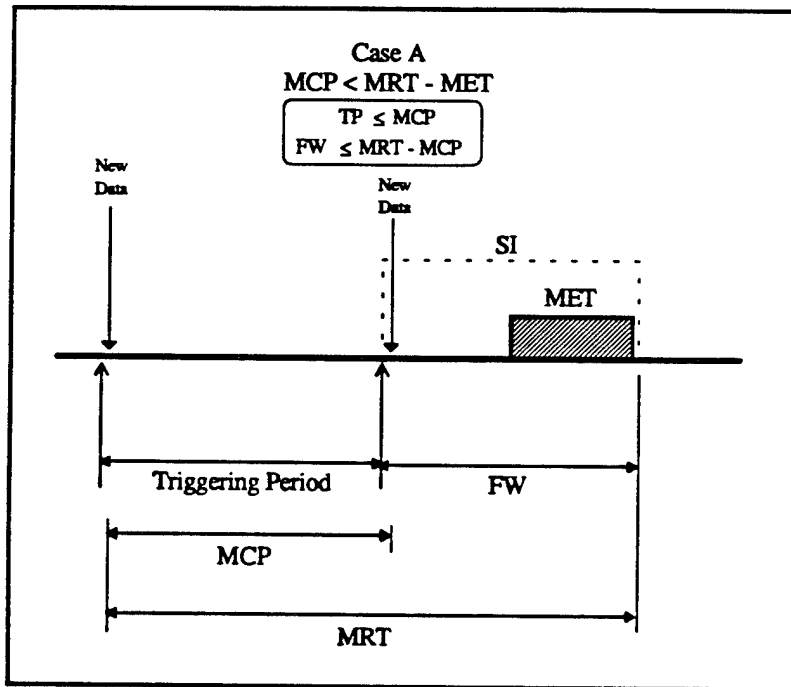
Figure 3.7. The Sporadic Conversion when MCP < MRT-MET

The second case, shown in Figure 3.8, occurs when MRT - MET ≤ MCP. This more constrained situation forces a further reduction in the triggering period. Thus, the new TP should be TP ≤ MRT - MET and the FW should be equal to MET.
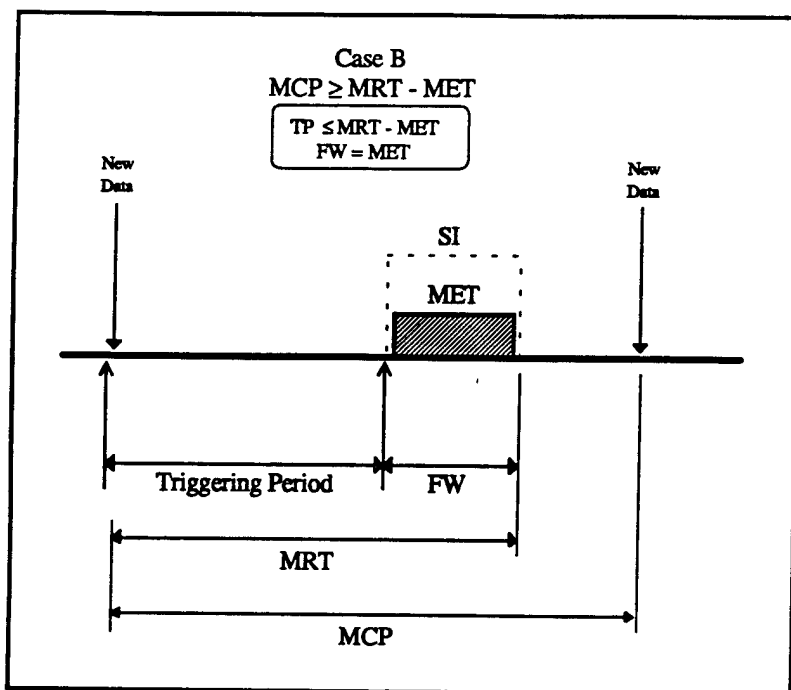
Figure 3.8. The Sporadic Conversion when MCP ≥ MRT-MET

In general, the triggering period should be

$$MET \leq TP \leq min(MRT - MET, MCP).$$

Nevertheless, in order to minimize the impact on the load factor of the prototype, it is desirable that TP be as large as possible, meaning that

$$TP = min(MRT - MET, MCP).$$

Now, assuming that the values for TP and FW have been established, so that the original timing constraints of the sporadic operator are satisfied, let's see what kind of relations should exist between the original values, so that we could validate them.

Clearly:

- MET ≤ MRT   (by Theorem 2)
- MET ≤ MCP   (by Theorem 1)   Eq. (1)
- MET ≤ TP   (by Theorem 1)

61

- $TP \leq MCP$ (for static scheduling)[4]

- $MET \leq FW \leq TP$ (Scheduling Model)   Eq. (2)

For case A: $MCP < MRT-MET$

$$TP = MCP$$ Eq. (3)

and

$$FW = MRT - MCP$$ Eq. (4)

Plugging (3) and (4) into (2),

$$MET \leq MRT - MCP \leq MCP$$ Eq. (5)

From the right inequality of (5),

$$MRT \leq 2 \times MCP$$

Plugging (1) into the left inequality of (5),

$$MRT \geq 2 \times MET$$

For case B: $MRT-MET \leq MCP$

$$TP = MRT - MET$$ Eq. (6)

and

$$FW = MET$$ Eq. (7)

Plugging (6) and (7) into (2),

$$MET \leq MET \leq MRT - MET$$ Eq. (8)

From the right inequality of (8),

$$MRT \geq 2 \times MET$$

Also,

$$MRT - MET \leq MCP \quad \text{or} \quad MRT - MCP \leq MET$$

Plugging (1) into the above inequality,

$$MRT - MCP \leq MCP \quad \text{or} \quad MRT \leq 2 \times MCP$$

Therefore the MRT for a sporadic operator must be upper bounded by twice its MCP and lower bounded by twice its MET, as follows:

---

[4] Otherwise we would have to be able to detect at run-time when new data had arrived, only possible with dynamic scheduling.

$$2 \times \text{MET} \leq \text{MRT} \leq 2 \times \text{MCP}$$

Note that when MRT assumes its lowest possible value, which is 2 × MET, the triggering period TP will also reflect its lowest possible value, which is MET, with FW still being equal to MET. This case is illustrated in Figure 3.9.
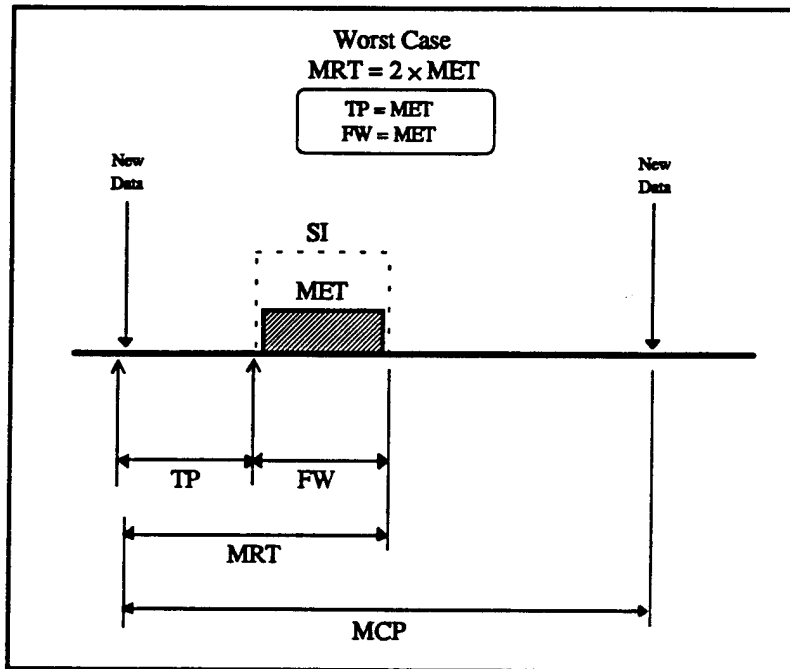
Figure 3.9. Worst Case Situation

Note that in both cases the conversion of a sporadic operator results in very stringent timing constraints to the equivalent periodic operator. This will definitely have a great impact on the schedulability of the prototype. In the second case, for example, there is no slack time for the converted operator, since FW = MET. This forces us to remove out portions of MET from the schedule, where no other operator could be scheduled.

Of course, the amount of slack time for this operator can be increased by decreasing its TP, but this will also increase the entire load factor. Basically, there exists a trade-off between load factor and slack time. How much to increase one in detriment of the other to increase schedulability is a very difficult question.

While this question does not have an answer, it does offer suggestions to help designers in finding solutions that best fit their needs.

When converting a sporadic operator into an equivalent periodic one, the triggering period (TP) can range from a minimum of MRT/2, where the slack time is equal to MRT/2 – MET, up to a maximum value equal to min(MRT-MET, MCP), implying that the slack time is max((MRT-MET-TP), 0).

First, define load factor contribution as LFC = $\dfrac{\text{MET}}{\text{TP}} - \dfrac{\text{MET}}{\text{TP}_{max}}$, i.e., the difference between the corresponding LF for a specific triggering period TP, and the load factor if TP were set to its maximum value. Within the interval MRT/2 $\leq$ TP$\leq$ min(MRT-MET, MCP), the slack time ST, which is the scheduling interval for the sporadic task minus its computation time, is defined as ST = MRT - MET - TP, as can be derived from Figures 3.7 and 3.8.

Clearly, when TP is maximum, the load factor contribution (LFC) is zero, in the sense that it cannot be increased any further. For the other values of TP, including those enforced in the conversions for the previous cases A and B, some considerations must be taken into account. Assume that MCP $\geq$ MRT-MET. Although it may appear at first that LFC varies with MRT, since TP is lower bounded by MRT/2, that is not the case, in other words, MRT only limits the valid range for TP. Figure 3.10 shows a family of curves for different values of MCP, and for a fixed value of MET and MRT. As explained earlier, LFC is insensitive to changes in MRT.

The load factor contribution LFC, as previously defined, is a function inversely proportional to the triggering period TP, and that it will decrease faster for periods less than $TP_c = \sqrt{\text{MET}}$, where its first derivative with respect to TP is equal to -1[5]. Note, however, that TP cannot be smaller than MET, meaning that TP_c will always be located

---

[5] Care must be taken to the fact that the derivative at some point being equal to -1, does not imply that the slope equals 135° at that point, since both axes may have different scales, as shown in Figure 3.10.

64

to the left of any valid value for TP. The main conclusion is that different values of MCP have very small effect in the variation of LFC. Similar conclusion can also be drawn for the case where MCP < MRT-MET. Therefore, in any case, the consequence is that we always have the full range of TP, from MRT/ 2, up to min (MRT-MET, MCP) to change TP, without causing any harm to the load factor of the system.
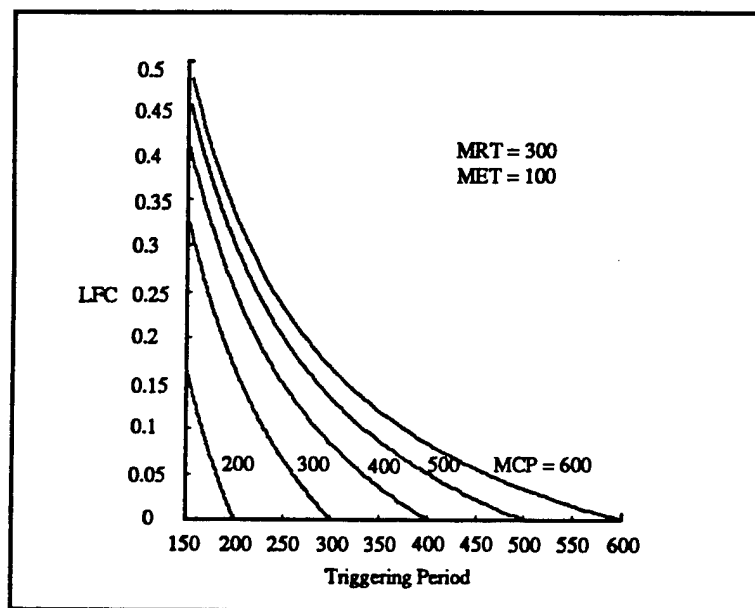
Figure 3.10. Effects of TP on the Load Factor

Note that the very first question remains unanswered, but now, the effects in the total load factor are more clearly understood when the triggering period is changed.

2. Important Remarks about the Conversion

This first idea of conversion of sporadic operators was introduced by Mok [Mok83] in his Lemma 2.3 which stated

> "Let $M = M_p \cup M_s$ be an instance of a process model. Suppose we replace every sporadic process $T_i = (c_i,p_i,d_i) \in M_s$ by a periodic process $T'_i = (c'_i,p'_i,d'_i)$ with $c'_i = c_i$, $p'_i = min(d_i-c_i+1, p_i)$ and $d'_i = c_i$. If the resulting set of all periodic processes M' can be successfully scheduled, then the original set of processes M can be scheduled without a priori knowledge of the request times of the sporadic processes in M_s."

65

Note, however, that although the idea of the transformation is valid, care must be taken to see the context in which that sporadic operator appears, since some of its attributes, such as minimum calling period, are totally dependent upon the producer of the triggering data and not on the sporadic operator itself. In other words, if the producer of data for some sporadic task is an external event that will be handled by some kind of interrupt handler, then there will be no influence whatsoever in the generation of the data, and the minimum period will be obeyed by the external device. However, if the producer is another task that will be included in our static schedule, it must be assured that two consecutive instances of the producer operator will not be scheduled closer than the minimum period specified for the sporadic consumer. In this case, the transformation alone is not enough, and an additional restriction must be imposed on the producer of the data. This situation is depicted in Figure 3.11.

In conclusion, it can be said that Mok's lemma by itself does not guarantee that a schedule really exists for the original set, even if the resulting set of all periodic processes M' can be successfully scheduled, unless as explained earlier, a restriction is imposed on the producers as well.
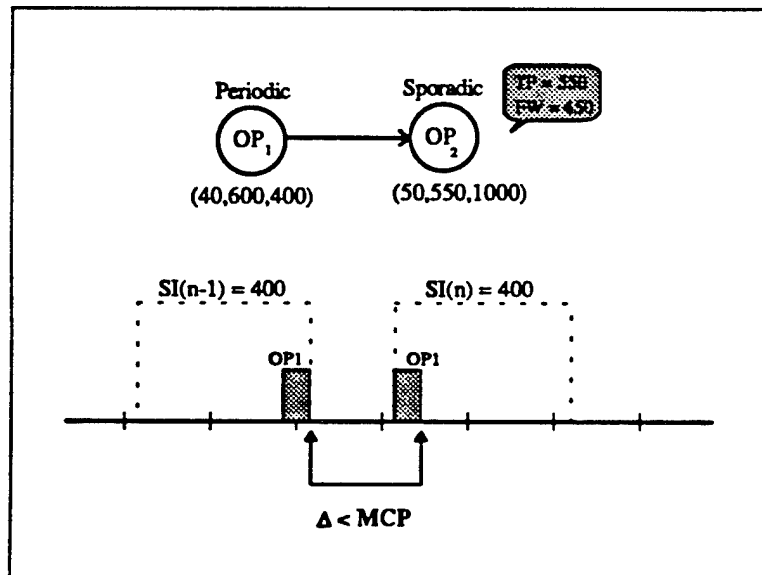
Figure 3.11. Restrictions on the Producer Imposed by the Consumer's MCP

3. Implementations Issues about the Conversion

When implementing this conversion it is strongly recommended that a careful analysis of the task graph be made to determine reasonable bounds for the period of the transformed sporadic operator. At first glance, an obvious upper-bound is the value of its MCP. However, for lower-bounds this choice is not so clear. Nonetheless, it is assumed that after this pre-processing there will be an interval of possible values for the period of the transformed sporadic task. The reason for these bounds is to provide us with some margin for making the conversion, so that the final harmonic block of the entire set is not increased significantly.

Given a set of sporadic operators, the following steps are suggested for the final choice of their periods:

1) Set the period of every sporadic task to its upper-bound, so that the total load factor is minimized

2) Try to find a feasible schedule for the entire prototype (if this is not possible pick one sporadic task)

3) Start decreasing its period;

4) For each new period check for schedulability;

5) Proceed until its lower-bound is reached. If no schedule is found reset its period to the upper-bound, pick another task and go back to step 3;

Another possible heuristic is to assign the smallest period among the periodic operators which is closest to but smaller than the upper-bound of the sporadic operator, and then proceed with the schedulability tests. One could also try to minimize the harmonic block. As can be seen, there are several possible heuristics, but there is no optimal solution. Nevertheless, it is understood that, due to the very stringent timing constraints resulting from the conversion, every possible attention should be given to this step.

IV. DISTRIBUTED SCHEDULING

A.   INTRODUCTION

For uniprocessor systems, most scheduling problems involving precedence constraints can be solved in polynomial time. Lawler [Law73] showed that scheduling non-preemptable tasks with unit computation times, deadlines, and arbitrary precedence constraints can be accomplished using the Latest Deadline First Algorithm in $O(n^2)$ time. Similar results were obtained by Lageweg, Lenstra, and Kan, even for tasks with an arbitrary computation time, if the release times were assumed to be zero for all tasks. Blazewicz [Bla76] proved that, for this scheduling problem, a preemptive schedule exists if and only if a non-preemptive schedule exists. Therefore, in this case, preemption need not be considered. Blazewicz also demonstrated that the Earliest Deadline First algorithm can also be used to schedule preemptable tasks. The only scheduling problem involving precedence relations that has been proven to be NP-complete is the non-preemptable case, where no restrictions are placed on the release times nor on the computation times. The non-preemptable case is also NP-complete if there are no precedence relations among the tasks [GJ77a].

Scheduling tasks with precedence constraints in multiprocessor systems is much more difficult than doing so in uniprocessor systems. For example, scheduling tasks with arbitrary precedence constraints and unit computation time is NP-hard both for the preemptive and the non-preemptive cases [Ull75, Ull76].

Many researchers have attempted to develop efficient heuristics algorithms to solve the general problem, but with limited success. In most cases, the researcher ended up restricting the solution space for specific cases, such as when the task graph is a forest, or when there are no precedence constraints.

In general, two different approaches to handling distributed computation can be identified. In the first, the distributed system is coordinated by a single system clock, which synchronizes all tasks so that computation progresses in a lock-step fashion, and

communication between tasks can only occur at specific times. In the second approach, tasks are synchronized only when necessary, and do so by executing appropriate handshake protocols. The former approach requires less inter-processor communication, but is rigid, and relies on a global clock whose implementation is by itself another very difficult problem to solve. The latter approach, although more flexible, dramatically increases the complexity of the synchronization problem, and may be very costly in terms of communication, since many acknowledge signals must be exchanged in order to maintain proper synchronization. The use of rigorous and more constrained timing requirements allows for the establishment of a weak form of synchronization among the tasks of the distributed system, and represents an alternative in the middle [Mok83].

B. ARCHITECTURAL ISSUES

This section is not intended to present an in-depth analysis of the effects of the architecture on distributed scheduling, but merely to introduce some of the problems so that the reader may be aware of their existence and importance.

In a distributed environment, it is very likely that one will have to deal with heterogeneous computers, each one with a different clock, different memory systems, and so forth. It is therefore important to realize how these attributes can affect scheduling.

1. Different Clocks

The precision of a clock is directly related to its granularity, the minimum number of ticks it can handle, and the quality of its time reference, which is usually based on some kind of crystal. The first limiting factor imposed by the clock, therefore, is the minimum acceptable period. This is not, however, an actual limitation, since typical clocks range from tens to hundreds of megahertz, providing an order of nanoseconds for the minimum allowable period. The real problem is that clocks can drift among themselves, causing a variety of synchronization problems. Maintaining an accurate global clock is one of the most challenging tasks in the distributed processing arena. Usually this is achieved at the cost of substantial overhead in communications.

2. Speed of CPUs

The net result when different processors are present is a different execution time for the same piece of code when running in the various processors. This factor necessitates previous knowledge of allocation by the scheduler, so that it can be taken into account. Within CAPS, this is accomplished automatically, because a kind of simulated time is used for scheduling, which is scaled according to the speed of the machine on which it runs.

3. Memory

Issues like cache size, paging, number of pipelining stages, etc., can affect the overall throughput of the system, and consequently the timing requirements, but hopefully all of these different delays are already taken into account by the specified maximum execution time of the task.

4. The Communication Media

This is one of the most important factors in dealing with distributed systems, and can greatly affect final timing requirements for the application. Note also that the timing requirements are affected not only by the actual transmission delay, but also by the operating systems functions invoked on behalf of the applications. In CAPS, for example, although there is a time-bounded protocol (FDDI) it is still necessary to make calls to the underlying Unix operating system, which has no support for real-time applications.

5. Interconnectivity

The number of processors, the distance by which they are separated, there abilities to communicate with one another, etc., are issues that should be raised before tackling the scheduling problem.

C. THE PROBLEM STATEMENT

To reiterate, the original objective of this research was to find better methods of supporting efficient and reliable scheduling of distributed hard real-time systems.

71

It is unquestionable that the ideal real-time distributed system should be able to support groups of tasks running asynchronously in different processors, each processor having its own internal clock. An additional goal, despite the precedence relations among the tasks, would be to eliminate the need for enforcement of any kind of synchronization required for communication. An even more important goal would be that all the deadlines and other requirements (such as no loss of data, etc.) could be met.

Being aware of the complexity of the message routing problem described in Chapter I and reviewing the alternatives presented in Section A, it appears to be that the best available option to achieve the ideal system is the very last alternative, i.e., to sacrifice timing constraints in order to decrease scheduling complexity. Unfortunately, that is not the current trend in most researches in the field of distributed scheduling today. Researchers are still trying to find better heuristics to scheduling algorithms so that the timing complexity for a sub-optimal case is decreased by some constant factor. But, due to the NP-Hard nature of the problem, it is most likely that some restrictions will be imposed on the initial problem.

This work moves in the other direction, in other words, investigating ways of restricting or relaxing the timing requirements so as to increase the chances of finding a feasible schedule. It is understood, however, that, depending on the application, this approach may not be practicable. It may well be that most of the timing requirements cannot be changed at all. However, this is most likely untrue for most cases. Especially in this applications framework, where the user is prototyping the intended system in the early stages of its life cycle, there is an opportunity to validate and change the system's requirements, which makes this approach very attractive. Note, however, that this discussion is not about missing deadlines or employing imprecise computations [LLS91], but focuses simply on relaxing timing constraints so that no synchronization is needed, and consequently decreasing substantially the complexity of the distributed scheduling problem.

The next section addresses the underlying semantics behind all possible combinations of triggering conditions, stream types and operator types within a valid PSDL program, so that later, when discussing the major synchronization issues, it is certain that all cases have been covered.

D. SYNCHRONIZATION IN PSDL

There are two kinds of streams in PSDL, Sampled Streams (SS) and Data Flow Streams (DF). Note, however, that within the former are two semantically different sub-types of streams, depending on the triggering condition of the consumer operator. If the consumer operator is not triggered (NT) by any data, then it should be understood that a specific data value can be lost or overwritten, or even read over and over again by the consumer, without any harm to the system. This type of behavior is very useful when reading sensor data. In most cases, the sensors will be able to generate data in a much higher rate than the consumer will read it, but the most recent data is of primary interest. Even for tracking systems, where the history of data values is very important, this kind of stream is still very useful. Note in Figure 4.1 that a specific value at some previous time t is not relevant, because the consumer is only interested in the average behavior, so that the filter algorithm can predict the future position of the target. In this kind of situation, no synchronization is needed, releasing the producer and consumer operators from any constraints on their periods.
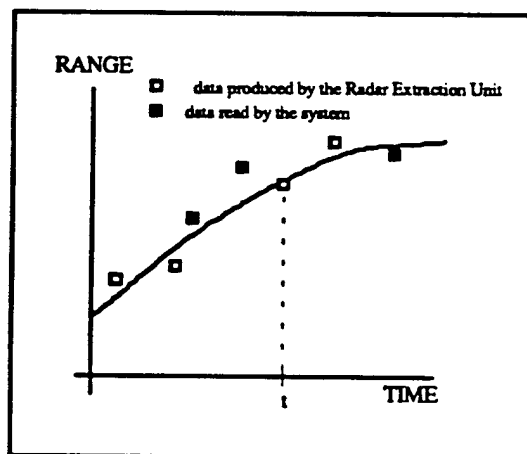
Figure 4.1. Typical Radar Data

73

The second type of Sampled Stream exists when the consumer operator is TRIGGERED BY SOME (TBS) data value. By definition, the consumer with this triggering condition should always catch a new piece of data if it is from one of the streams specified in the TRIGGERED BY SOME clause. For example, if some operator OP1 is TRIGGERED BY SOME X, Y, then, if new data is coming from either X or Y, it should be guaranteed to be read, and not lost or overwritten.

Although buffer overflow or underflow is not an issue, due to the way sampled streams are defined, the only way to avoid loss of data in this case is to enforce the condition that $PER_{producer} \geq PER_{consumer}$, and, consequently, the synchronization problem will have to be handled accordingly.

Finally, in the case of Data Flow Streams, where the consumer is TRIGGERED BY ALL, the inputs specified in the TRIGGERED BY ALL clause for new data should be examined, and if all of them happen to have new data in their buffer, they should be consumed, firing the operator. The TRIGGERED BY ALL condition can be thought of as being a logical AND among the streams declared in the TRIGGERED BY ALL clause. Clearly, in this case, there is also a need to enforce $PER_{producer} \geq PER_{consumer}$ so that no data is lost, and once again the synchronization problem must be handled explicitly.

The basic semantic difference between the TRIGGERED BY ALL data flow streams and the TRIGGERED BY SOME sampled streams is that if for any reason the data is not consumed and another piece of new data arrives, in the former it will raise a buffer overflow exception, while in the latter the data will be simply overwritten.

E.   DEALING WITH SPECIAL CASES

Data flow streams are currently implemented in CAPS as a FIFO queue of buffer size one. This imposes an important restriction on the PSDL program, that is, all producers of data flow streams to some unique consumer should have the same period, or a FIFO buffer overflow may occur in one of the streams, even if the condition $PER_{producer} > PER_{consumer}$ is satisfied (Figure 4.2). This happens because OP1 may

write twice before OP2 outputs some value so that the triggering condition can be satisfied. This problem usually reflects a possible design error, because it makes no sense to have an operator being triggered simultaneously by two data events that are produced with different rates. A possible and recommended solution is to force all producers of data flow streams to a unique consumer to have the same period.
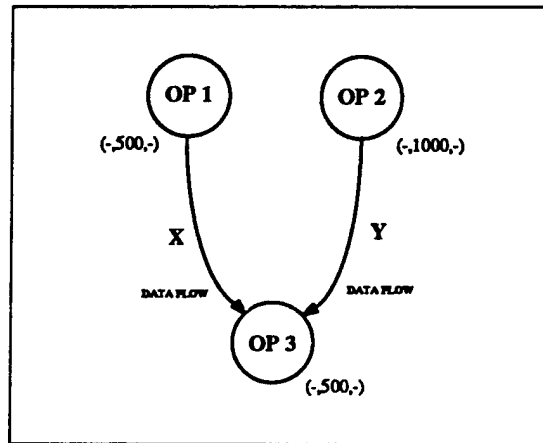
Figure 4.2. Producers with Different Periods

Another important issue is that, although it is semantically correct in PSDL to have several operators writing to the same data flow stream, or even to the same TRIGGERED BY SOME sampled stream, as illustrated in Figure 4.3, this case cannot be handled unless an upper-bound is placed on the number of concurrent copies of a stream in a PSDL program. This restriction is due to the fact that streams have limited buffer size, and if the number of copies is very large there is no way to guarantee that one operator will not write to the stream right after the other, and therefore cause an overflow. In the uniprocessor case, the only way to handle this problem is by imposing very hard restrictions on the period of the consumers, so that it will be limited to, at most, half of the minimum MET of the producers. This result may be seen as an extrapolation to this case of Nyquist's well known sampling period theorem. Currently, CAPS does not enforce this condition.

75
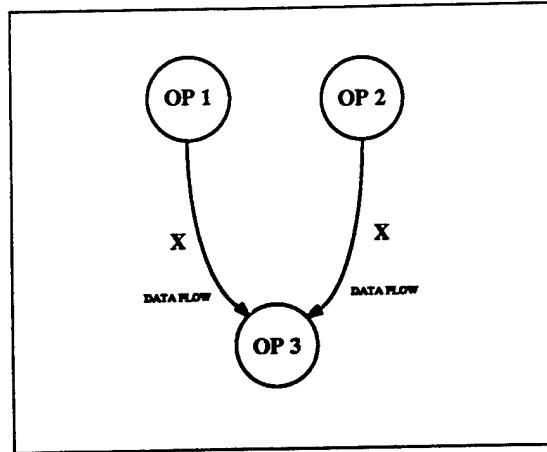
Figure 4.3. Potential Overflow Situation

Still, due to the powerful semantics of PSDL, there is another problem to solve, which is the possibility of the same stream being data flow for some consumers and sampled stream for others, as illustrated in Figure 4.4. To make things worse, these streams can even have different latencies.
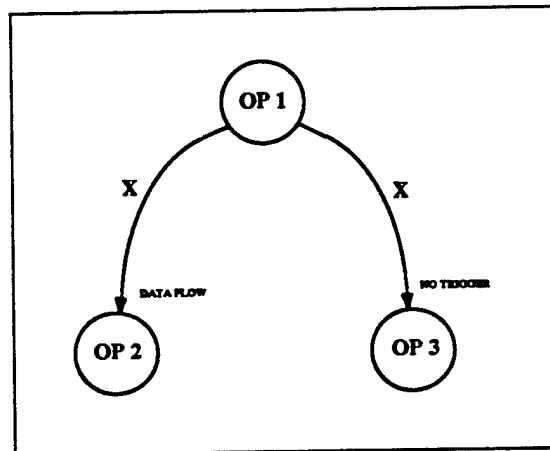
Figure 4.4. Different Stream Types Combination

Actually, there are some other cases that could also be cleverly checked, so that users could receive some suggestions and warnings about their design, like for example in the case illustrated in Figure 4.5, where OP_1 could have its period increased and consequently lowering the load factor, since it will not do any good to keep its period smaller than OP_2.
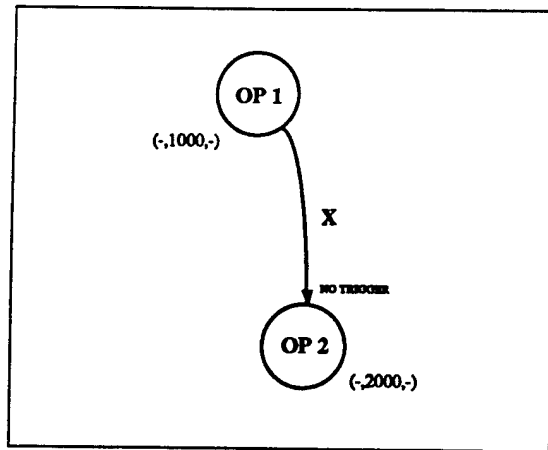
Figure 4.5. Period Incompatibility among Operators

As one can expect, the above cases make the validation process of a PSDL program very complex. For the sake of completeness, the semantic checks and stream type derivations for all possible combinations of operator types and data triggering conditions in PSDL are listed in Table 4.1. The actions which should be taken by the scheduler for each one of those possible combinations will also be presented.
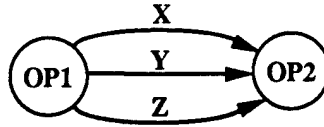
Type	OP1	OP2	Data Trigger	X	Y	Z	Action/Check	OBS
TC-TC	P	P	By All X,Y	DF	DF	SS	If $P_{OP1} \leq P_{OP2}$ then Error	1
	P	P	By Some X,Y	SS	SS	SS	If $P_{OP1} \leq P_{OP2}$ then Error	2
	P	P	None	SS	SS	SS	$P_{OP2} = \max(P_{OP1}, P_{OP2})$	
	P	S	By All X,Y	DF	DF	SS	OP2.upper = min(OP2.upper,P)	1,3
	P	S	By Some X,Y	SS	SS	SS	OP2.upper = min(OP2.upper,P)	2,3
	P	S	None	SS	SS	SS	Error: Cannot be Sporadic	5
	S	P	By All X,Y	DF	DF	SS	OP1.lower = max(OP1.lower,P)	1,3
	S	P	By Some X,Y	SS	SS	SS	OP1.lower = max(OP1.lower,P)	2,3
	S	P	None	SS	SS	SS		5
	S	S	By All X,Y	DF	DF	SS	OP1.actual ≥ OP2.actual	1,4
	S	S	By Some X,Y	SS	SS	SS	OP1.actual ≥ OP2.actual	2,4
	S	S	None	SS	SS	SS	Error: Cannot be Sporadic	5
TC-NTC	P	NTC	By All X,Y	DF	DF	SS	Error: Cannot be Data Flow	1
	P	NTC	By Some X,Y	SS	SS	SS	Error: Possible Data Loss	2
	P	NTC	None	SS	SS	SS		
	S	NTC	By All X,Y	DF	DF	SS	Error: Cannot be Data Flow	1
	S	NTC	By Some X,Y	SS	SS	SS	Error: Possible Data Loss	2
	S	NTC	None	SS	SS	SS		
NTC-TC	NTC	P	By All X,Y	DF	DF	SS	Warning: Possible Overflow	1,6
	NTC	P	By Some X,Y	SS	SS	SS	Warning: Possible Data Loss	2
	NTC	P	None	SS	SS	SS		
	NTC	S	By All X,Y	DF	DF	SS	Warning: Possible Overflow	1,6
	NTC	S	By Some X,Y	SS	SS	SS	Warning: Possible Data Loss	2
	NTC	S	None	SS	SS	SS	Error: Cannot be Sporadic	5
NTC-NTC	NTC	NTC	By All X,Y	DF	DF	SS		1,7
	NTC	NTC	By Some X,Y	SS	SS	SS		2,7
	NTC	NTC	None	SS	SS	SS		

Table 4.1. PSDL Data Triggering Semantic Table

LEGEND

TC = Time-Critical Operator	P = Periodic Operator/Period	SS = Sampled Stream
NTC = Non-Time-Critical Operator	S = Sporadic Operator	DF = Data Flow Stream

In Table 4.1, "upper" and "lower" represent, respectively, the maximum and the minimum values the equivalent period of the sporadic operator can assume. They are initially set, respectively, to infinite and zero. "Actual" is the value of the triggering period

of the sporadic operator after the conversion is done. As can be seen in Table 4.1, in all TRIGGERED BY ALL cases it is necessary to prevent, or at least give warnings, whenever the producer operator is faster than the consumer, so that no loss of data or overflow will be incurred [Table 4.1(1)]. Similarly, in the TRIGGERED BY SOME cases, this constraint must also be enforced, but in this case the motivation is to prevent loss of data, since Sampled Streams, by definition, do not overflow [Table 4.1(2)].

When dealing with sporadic operators upper and lower bounds are defined for their triggering periods, so that later, when conversion of the sporadic operators to equivalent periodic operators takes place, it is certain that all of these constraints are taken into consideration [see Table 4.1(3)]. The sporadic to sporadic case (S-S) cannot yet be handled with upper and lower bounds, since there can be up to five different possible overlapping patterns for their period interval. Hence, final checking of this case will be delayed until the equivalent periods have been calculated [Table 4.1(4)].

Another important point to mention is that consumers with no data triggering condition must be periodic, or an error will be raised [Table 4.1(5)].

Finally, although very unlikely to happen, it should be pointed out that it may happen, for unexpected reasons, such as a lot of slack time left over from the static scheduler, that some non-time-critical operator may be fired more than once in the same Harmonic Block, leading to a possible overflow if they are connected by data flow streams to time-critical operators [Table 4.1(6)]. This is not a concern among NTCs, since all of them will be executed consecutively, in other words, between two consecutive instances of any NTC operator is guaranteed to have an instance of all the remaining ones [Table 4.1(7)].

Table 4.2 presents all possible combinations of the PSDL timing constraints and the resulting actions and checks to be performed by the scheduler.

MET	MRT	MCP	PER	FW	OPERATOR TYPE	ACTIONS/CHECKS
N	N	N	N	N	NTC	
"	"	"	N	S		ERROR
"	"	"	S	N		ERROR
"	"	"	S	S		ERROR
"	N	S	N	N		ERROR
"	"	"	N	S		ERROR
"	"	"	S	N		ERROR
"	"	"	S	S		ERROR
"	S	N	N	N		ERROR
"	"	"	N	S		ERROR
"	"	"	S	N		ERROR
"	"	"	S	S		ERROR
"	S	S	N	N		ERROR
"	"	"	N	S		ERROR
"	"	"	S	N		ERROR
"	"	"	S	S		ERROR
S	N	N	N	N	SPORADIC	Auto-Pick MRT and MCP
"	"	"	N	S		ERROR
"	"	"	S	N	PERIODIC	FW = PER
"	"	"	S	S	PERIODIC	MET ≤ FW ≤ PER
"	N	S	N	N	SPORADIC	MCP≥MET; MRT=MET+MCP
"	"	"	N	S		ERROR
"	"	"	S	N		ERROR
"	"	"	S	S		ERROR
"	S	N	N	N	SPORADIC	MRT≥MET; MCP=MRT
"	"	"	N	S		ERROR
"	"	"	S	N		ERROR
"	"	"	S	S		ERROR
"	S	S	N	N	SPORADIC	MET≤MCP; MET≤MRT
"	"	"	N	S		ERROR
"	"	"	S	N		ERROR
"	"	"	S	S		ERROR

Table 4.2. PSDL Timing Constraints Semantic Table

LEGEND
N = Not Supplied
S = Supplied

Table 4.2 shows that very few combinations of PSDL timing constraints are semantically acceptable. The only one that deserves some explanation is the case where only the MET is supplied. In this case, the scheduler picks up a pair of values for MCP and MRT, so that the individual load factor of the sporadic operator is equal to

$$\frac{\max((0.75 - \sum LF_{PER}), 0.1)}{\text{\# of sporadic operators}}$$

This approach relieves the designer from having to define timing constraints for sporadic operators, which might not be clear yet, at that stage of the prototyping, and it also tries to decrease the timing requirements for that sporadic operator. However, it is dangerous, in the sense that it will always increase the load factor of the prototype to at least 0.75, even if the total load factor for all periodic operators was very low.

As is apparent, most of the semantic checks, mainly those related to the control constraints part of the PSDL program, such as data triggering checks and timing constraints checks, are left up to the scheduler to implement. It is proposed that in the future CAPS releases some of these checks are taken from the scheduler and inserted into the Syntax Directed Editor (SDE), so that the user is not allowed to proceed to the translation step until he has a valid PSDL program. In doing so, the designer will not have to come all the way back to SDE if a semantic error is found.

F.   TACKLING THE SYNCHRONIZATION PROBLEM

It is clear that the most important issues in dealing with synchronization are the periods of producer and consumer tasks. However, even in the uniprocessor case, with the period of the consumer being smaller than the period of the producer, it can be easily shown that the synchronization is not always a good alternative. Figure 4.6 shows an example where no feasible schedule exist if synchronization is enforced, but it does exist otherwise. Three outcomes are possible if the synchronization is not required. First, if the consumer operator is TRIGGERED BY ALL X,Y , the proposed schedule is valid but X and Y will be consumed one instance later. If it is TRIGGERED BY SOME X,Y , then the schedule is always valid, because X and Y do not need to be consumed together. Finally, if there is no trigger, then the relative order is not important anyhow.

81
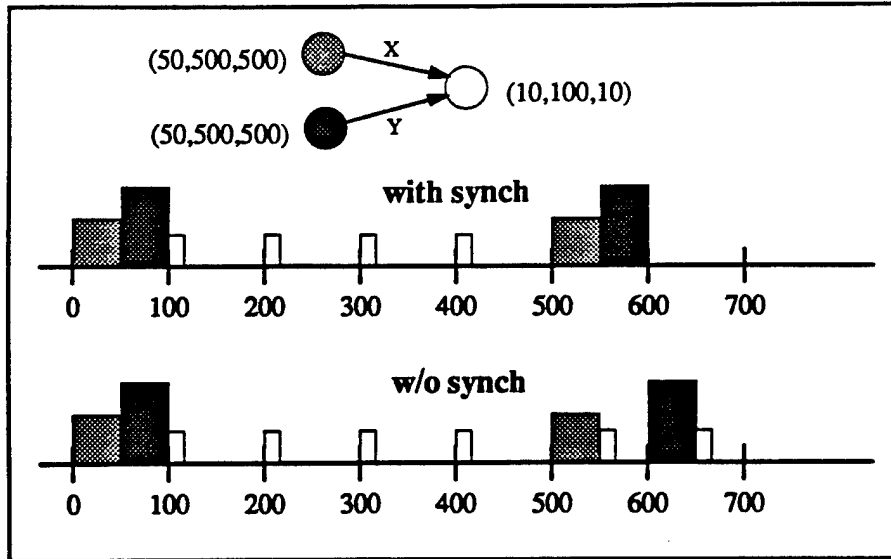
Figure 4.6. Reason for No Synch when $PER_{prod} \geq PER_{cons}$ (Uniprocessor Case)

From another perspective, if $PER_{producer} < PER_{consumer}$, then the streams connecting them should be sampled streams, because otherwise the data flow streams would overflow. Since the loss of data is possible("possible" because the data might well not be produced at all) the consumer cannot be TRIGGERED BY SOME either.

The only case in which $PER_{producer} < PER_{consumer}$ can be allowed is when there is no trigger at all. In this situation, synchronization is not needed, since it would place one additional burden on the scheduler, and would not solve the problem of loosing data. The only advantage to having synchronization points in this case is the fact that there would be a fixed pattern for losing data. Furthermore, by not having explicit synchronization, the most that could happen is that the consumer operator would read either the previous or the next instance of the data output by the producer, in other words, at most one producer period apart.
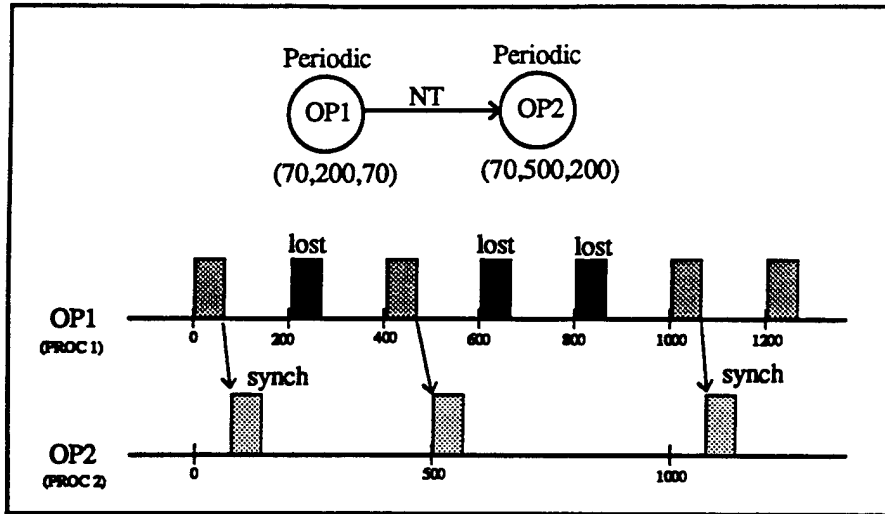
Figure 4.7. Reason for No Synch when $PER_{prod} < PER_{cons}$ (Distr. Case)

The second possibility is $PER_{producer} \geq PER_{consumer}$. In this case, the synchronization also does not solve the problem, since it is possible to have two instances of the producer operator being scheduled, one after the other, causing overflow or loss of data depending on the triggering condition. This case is illustrated in Figure 4.8.
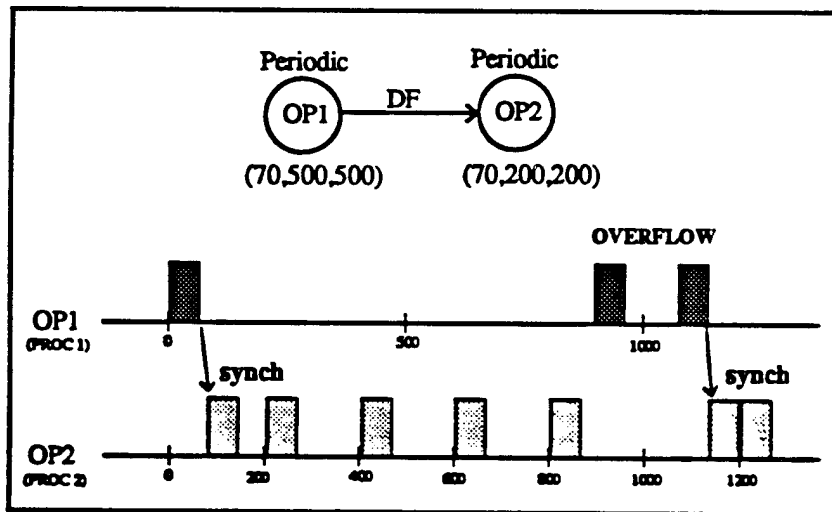
Figure 4.8. Reason for No Synch when $PER_{prod} \geq PER_{cons}$ (Distr. Case)

At first, one may conjecture that no synchronization is needed when $PER_{producer} \geq PER_{consumer}$, since it would be possible to catch every single occurrence of data ever

83

produced. However, this conjecture is untrue, due to the fact that the periodic input is not periodic in the common sense that is understood in electrical engineering and other related fields, as a pulse that occurs every t units of time!
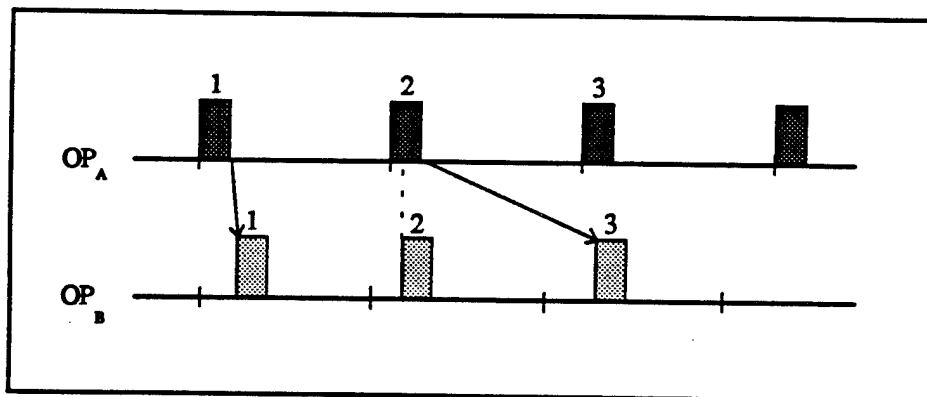
Figure 4.9. Synchronization among Periodic Operators when $FW_A = MET_A$

If that was so, the period ratio among producer and consumer would be a necessary and sufficient condition for guaranteeing synchronization, according to the following argument:

Assuming that $PER_B \leq PER_A$ (Eq. (1)) and that the phase of operator A is zero, there could be two cases:

1st case: start of second instance of B is less than finish of second instance of A

$$s_{2B} < f_{2A}.$$
Eq. (2)

In this case B just lost A, and therefore it is necessary to prove that the third instance of B will certainly catch the second instance of A. Formally

$$s_{3B} < f_{3A}$$

By the definition of periodic operator, and from Eq. (1),

$$s_{2B} < s_{3A}$$

But also,

$$s_{3B} = s_{2B} + PER_B$$
Eq. (3)

and

$$s_{3A} = s_{2A} + PER_A \quad \text{or} \quad f_{3A} = f_{2A} + PER_A$$
Eq. (4)

84

Plugging equations (1) and (2) into (3),

$$s_{3B} < f_{2A} + PER_A \qquad\qquad Eq. (5)$$

Finally, combining (4) and (5),

$$s_{3B} < f_{3A} \qquad\qquad \square$$

2nd. case: $s_{2B} > f_{2A}$. Trivial case where the second instance of B will catch the second instance of A. $\qquad\qquad\square$

In general, $s_{iB} < s_{iA}$ implies $s_{(i+1)B} < s_{(i+1)B}$ and hence, neither loss of data or buffer overflow can happen.

However, as explained before, this periodic definition is slightly different, in the sense that it may occur anywhere inside the period slot, invalidating our previous argument.

Within this framework, things are made much more complex, and the synchronization approach needs to change considerably.

The key question to be answered is: *What is the real need for synchronization between two operators, and when is it applicable?* As shown in the previous examples, the synchronization is not solving the problem and it is placing an additional burden on the scheduler.

Other question to be asked is:

Can every single piece of data coming from both data flow streams and from TRIGGERED BY SOME sampled streams be guaranteed to be consumed in a timely fashion, so that no overflow or loss of data occurs?

The answer is clearly yes, if after scheduling each producer of a data flow or TRIGGERED BY SOME sample stream, the consumer of that data flow stream, or of that sampled stream, is scheduled before the next instance of the producer.

In a uniprocessor case, or even in a shared memory multiprocessor model, this approach is acceptable and easy to implement and guarantee. This, by the way, is how it is implemented right now in CAPS. However, in a truly distributed case, besides the

difficulty in implementing this approach, the lack of a master clock might cause a feasible schedule to become unfeasible. This assertion may be illustrated with a simple example. Assume a schedule for a two-processor system that meets all deadlines and synchronization requirements among their tasks, and that no buffer overflow occurs with respect to the data flow streams. Now, if clock drift occurs in processor 2, so that one of its consumers gets shifted more than twice the period of its correspondent data flow producer, the consumer is guaranteed to lose data, and the schedule will fail.

Therefore, although the uniprocessor and the shared-memory multiprocessor cases can be handled appropriately, a new approach must be developed for the distributed case. Ideally, several sets of communicating processes would run independently in each processor, but with the guarantee that no data would be lost and no deadlines missed.

It will be useful to review the synchronization problem between producers and consumers. What is the real meaning of missing a deadline within the context of a real-time system? It means that some process did not generate its output within the specified amount of time, and therefore the consumer could not consume the data, and so on. What is important here is that missing deadlines are always attached to data not being generated or consumed in the proper timing, and this is going to be the key-point in the approach, i.e., attempting to guarantee that all data being generated is consumed in a timely fashion.

Clearly, the very first condition that must be satisfied is that $PER_{producer} \geq PER_{consumer}$ so that no data is lost. It also seems obvious at first, that the worst case that can ever happen is when two consecutive instances of the producer are fired one after the other, and the consumer is scheduled about two periods apart. Unfortunately this is not true, as illustrated by the following Figure 4.9.
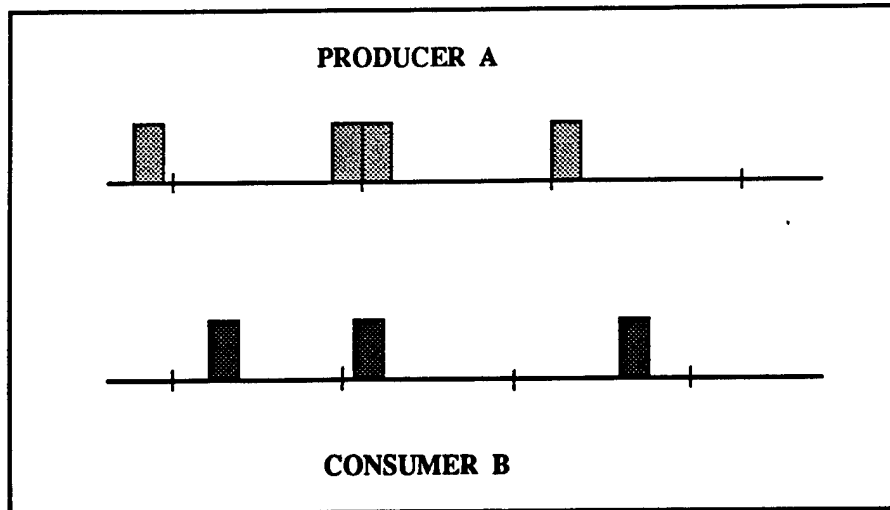
Figure 4.10. The Consumer-Producer Paradigm

Figure 4.10 shows that even with a faster consumer ($PER_B \leq PER_A$) one cannot discard the possibility of having more than one, actually even three occurrences of the slower producer between two consecutive instances of the consumer. This finding raises the following additional questions:

1) Under what conditions could that happen?

2) Is there an upper-bound on the number of instances of producers between two consecutive instances of the consumer? What would it be?

To answer these questions, analyze carefully Figure 4.10.

By construction:

$$PER_A + 2 \times MET_A \leq 2 \times PER_B \qquad \text{Eq. (1)}$$

and

$$PER_B \leq PER_A \qquad \text{(Initial Assumption)}$$

By definition of periodic operator

$$0 \leq MET_A \leq PER_A$$

By re-arranging Eq. (1)

$$MET_A \leq PER_B - \frac{PER_A}{2}$$

Thus, PER_B must be $\geq \dfrac{PER_A}{2}$ or otherwise MET_A would have to be negative.

Therefore we end up with the following solution interval for PER_B:

$$\frac{PER_A}{2} \leq PER_B \leq PER_A$$

and consequently

$$0 \leq MET_A \leq \frac{PER_A}{2}$$

The above inequality answers the first question by showing under what conditions

the situation depicted in Figure 4.10 can happen, i.e., whenever $MET_A \leq \dfrac{PER_A}{2}$.

To answer the second question, let us assume the situation presented in Figure 4.11, where four instances of the producer are attempting to exist in between the same two instances of the consumer.
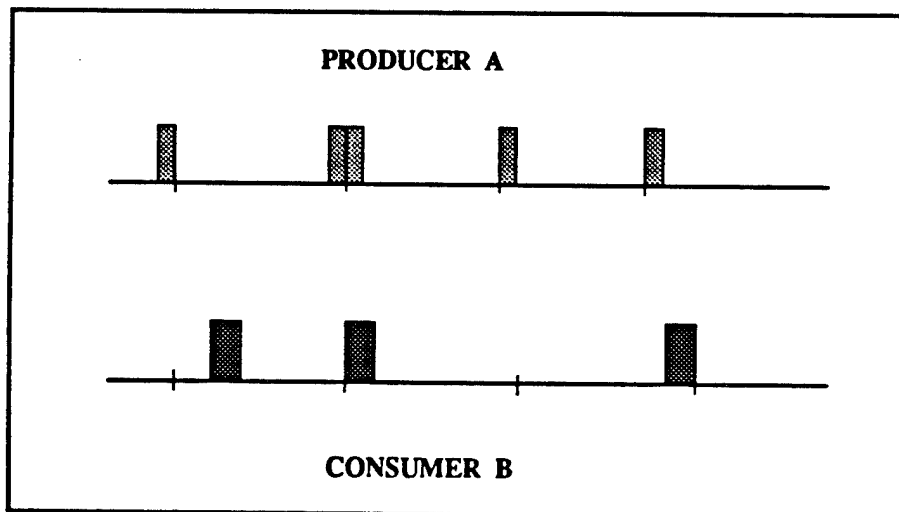
Figure 4.11. Seeking for an Upper-Bound

Eq. (1) now becomes

$$2 \times PER_A + 2 \times MET_A \leq 2 \times PER_B$$

Now let $MET_A = 0$, which is the best case possible. This results in $PER_B > PER_A$. But then there is no solution for the set of inequalities, i.e., three is actually the upper-bound.

Based on these results the following lemmas can be stated:

Lemma 1:

"Given a pair of operators, where one is a producer and the other is a consumer, and assuming that the period of the producer is bigger than the period of the consumer, there can exist at most three instances of produced data waiting to be consumed at any instant of time".

Lemma 2:

"Any produced data will be consumed within at most two periods of the consumer".

Finally, these lemmas allow the **Fundamental Synchronization Theorem**, that will be most useful in the distributed case, but that can be applied as well in the uniprocessor case.

Theorem 9:

"If there exists a feasible schedule that runs without buffer overflow or loss of data in a shared memory multiprocessor model, then there can be a distributed and totally independent schedule, without any kind of explicit synchronization, if the buffer size of the data flow streams, as well as for the sampled streams with a triggered by some condition have a size of three."

1. Additional Restrictions Imposed on the Timing Constraints

Obviously, a price is paid for getting rid of the synchronization, and it is reflected in a more stringent set of timing constraints for tasks.

Looking back at Figure 4.10 it can be seen that the worst case that can happen is to have some data from a producer consumed after $2 \times PER_B - MET_B$ units of time.

Currently, in PSDL, contrary from the sporadic case, there is no upper-bound on the time an input data for a periodic operator should be consumed. So, if the consumer is

a periodic operator that receives data from network streams, the fact of not using synchronization, will not impose any additional constraints on their timing requirements.

In the sporadic case however, the explicit upper-bound for consuming the incoming data is its MRT, which is assumed to be greater than or equal to the latency plus the MET of the consumer operator for the incoming data. Therefore, an additional restriction on the triggering period of a sporadic operator must be imposed when it has any data coming from network streams.
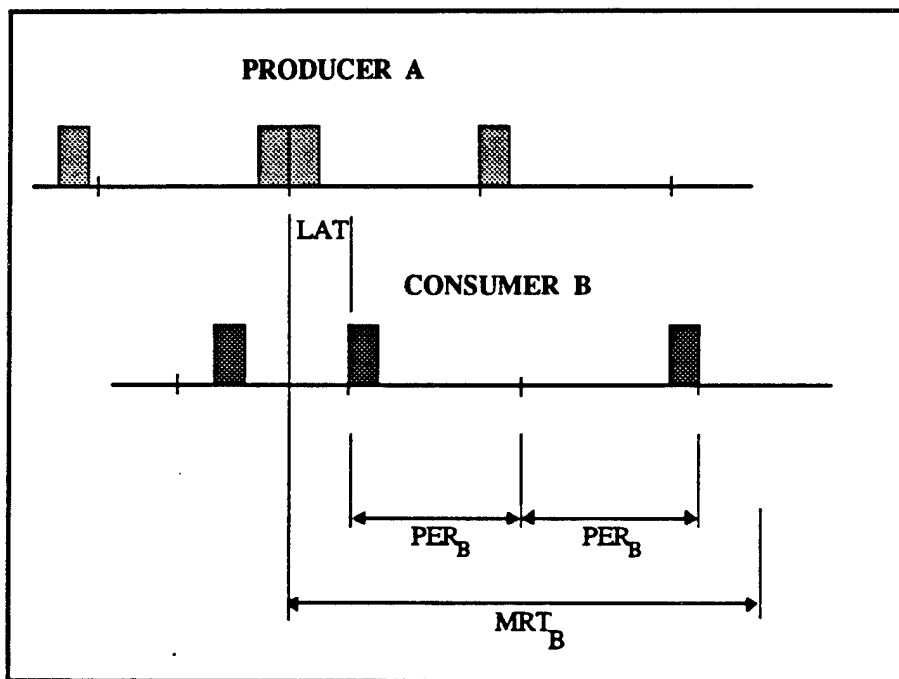
Figure 4.12. New Timing Constraints for the Sporadic Operator

From Figure 4.12

$$2 \times TP_B + LAT_{MAX} \leq MRT_B$$

or

$$TP_B \leq \frac{MRT_B}{2} - \frac{LAT_{MAX}}{2}$$

which is the new upper-bound for the triggering period of a sporadic operator.

From Chapter III, Section E, it is also know that $TP \geq MET$. Hence,

$$\text{MET}_B \le \text{TP}_B \le \frac{\text{MRT}_B}{2} - \frac{\text{LAT}_{MAX}}{2}$$

which is the new formula for calculating the triggering period of a sporadic operator, under the no synchronization assumption.

G.   THE TASK ALLOCATION MODEL

Two basic and unavoidable steps when designing distributed software systems are the decomposition of the system functions into software processes during the early stages of the design and, later on, the allocation of these processes to the several processors. Although sometimes these two steps are used interchangeably, they are very different activities.

Given the software requirements, the designer must first identify a set of logical interrelated modules and perform its functional decomposition. This can be done with the aid of traditional design methods, such as structured and object oriented design. For real-time systems, such decomposition will require consideration of critical timing constraints and may require introduction of special modules for synchronization [SW89].

The first major activity is partitioning, which is the mapping of these logical modules into a set of physical processes. The second is allocation (sometimes called assignment) which is the mapping of each process to one or more processors. The focus of this chapter is on allocation; for further reading on partitioning see Shatz and Wang [SW89].

As shall be seen, task allocation dramatically complicates the already complex problem of distributed software design, because assigning m processes onto n processors, there are n^m different possible assignments. Optimal allocation is a problem of exponential complexity, and it was proven to be NP-complete by Mok [Mok83].

The key to process allocation is to establish an allocation model in terms of a cost function and additional constraints that match the application requirements as far as logical and timing correctness. The goal is to minimize the cost function under the constraints.

Most of the cost functions found in available literature deal with performance. Others, such as those relating to reliability and fault-tolerance, are only now emerging [SW89].

The most widely used performance cost functions are:

1) *Interprocessor communication cost (IPC)* which is a function of the amount of data transferred, the network topology and link capacity;

2) *Load balancing*, which is a measure of how uniform the workload among the processors is. A good load balancing will maximize the system stability, which is the capability of busy hosts to receive bursty arrivals of processes without compromising their deadlines.

3) *Completion time*, the total execution time including interprocessor communication incurred by that processor.

The most frequent constraints found in typical real-time systems are due to hardware limitations of some processors, dependence of some processes on certain processors, and number of available processors.

The choice of a cost function obviously depends on the application, on the underlying hardware, and on several other characteristics.

Although distributed processing seems very attractive, one should be aware of the *saturation effect* (Figure 4.13) that is sometimes forgotten by many developers. The basic consequence of this effect is that, contrary to expectations, the throughput doesn't increase linearly as the number of processors is increased. Actually, at some point (which can be as few as three or four processors) throughput actually starts to decrease. Examples of this phenomenon are documented by Chu, et al. [CHL80] and by Jenny [Jen77]. The decrease in throughput is due to the excessive interprocessor communication, which is similar to the trashing problem in the early memory paging systems.
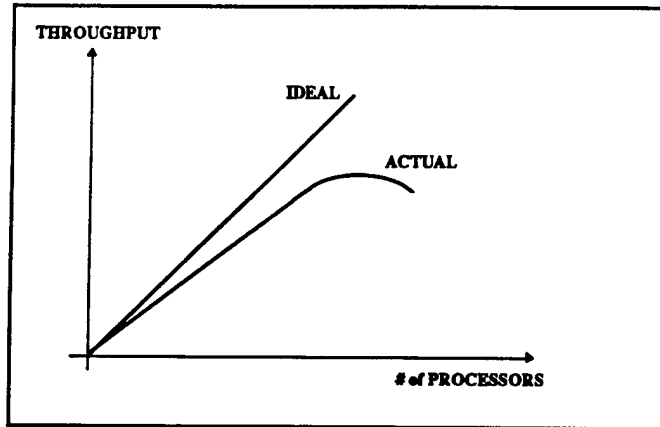
Figure 4.13. The Saturation Effect

Basically, all of the different approaches to solve the allocation problem fall into one of the three major classification areas: graph theoretic, mathematical programming, or heuristic methods, which are by no means mutually exclusive.

The first of these represents the processes to be allocated as nodes in a graph, where each edge has a weight that is proportional to its inter-module communication cost (IMC), with the following remarks: an IMC of zero means that no communication takes place between those two modules and an IMC of infinity means that they should be assigned to the same processor. If a minimal-cut algorithm is performed on the graph one ends up with the minimum allocation cost for those modules between two processors. In general, however, an extension of this method to an arbitrary number of processors requires an n-dimensional min-cut flow algorithm, which quickly becomes computationally intractable.

The mathematical programming approach uses, in most cases, the non-linear integer programming technique, where the above problem is formulated as a set of equations. It is very flexible in the sense that additional constraints can be included in the model very easily, however it has two short-comings. First, it fails to accurately represent real-time constraints and precedence relations among the tasks, because both factors introduce queuing delays into the system in a complex manner [DSWE83].

Finally, the heuristic methods, unlike the first two, try to find sub-optimal solutions for the assignment problem, which are in general faster, more extendible and simpler.

1.   Some Basic Definitions

Defining several metrics will provide a better insight into the problem.

Average Task MET - given n tasks, it is a lower-bound in the response time;

$$MET_{AVG} = \frac{\sum MET}{n}$$

Average Load Factor - it is a kind of schedulability index that shows how tight the system is. The bigger it is the harder is to find a schedule. It is independent of the number of processors, e.g., $LF_{AVG} = 0.8$ means that almost every operator is very CPU-intensive. A more precise insight could be obtained by the standard deviation of the load factor.

$$LF_{TOT} = \sum \frac{MET}{PER}$$

$$LF_{AVG} = \frac{LF_{TOT}}{n}$$

Average Processor Load Factor - given the number of processors p, it specifies the ideal share of processing so that a perfect load balancing is achieved.

$$PLF_{AVG} = \sum \frac{LF_{TOT}}{p}$$

Maximum Processor Load Factor - it specifies the maximum load factor each processor can sustain using the minimum number of processors.

$$PLF_{MAX} = \frac{\sum LF_{TOT}}{\lceil \sum LF_{TOT} \rceil}$$

Placement Cost Matrix - it basically shows the cost incurred when operator X is allocated to processor k. If some task must be placed in some specific processor, its placement cost should be zero. Otherwise it should be infinity. Other values reflecting the user's desires can also be used so that the scheduler will have more options when deciding upon the allocation.

Placement Cost	Processor 1	Processor 2	Processor 3
Operator A	∞	0	4
Operator B	0	∞	7
Operator C	5	8	5

Table 4.3. Placement Cost Matrix

Inter-Module Communication Cost Matrix - it basically shows the cost incurred when operator X wants to communicate with operator Y, or vice-versa, using the network. Note that it should be symmetric, since it doesn't depend on the way the communication is carried out. It simply states that if those two operators are allocated in different processors, that will be the amount of communication they will have to exchange. In this case it will also account for the state streams.

IMC Cost[1]	Operator A	Operator B	Operator C
Operator A	-	7	13
Operator B	7	-	8
Operator C	13	8	-

Table 4.4. IMC Cost Matrix

Distance Cost Matrix - it takes into account the geographic distance between processors. For all distances within a local area network, index 1 is assumed. When not connected, the distance is assumed to be infinite. If passage through additional networks is required, there will be an increase of 0.1 for each additional level of networking. Note that the basic purpose of this matrix is to see if the specified latencies and network delays are compatible with the underlying hardware architecture.

Distance Cost	Processor 1	Processor 2	Processor 3
Processor 1	0	1	∞
Processor 2	1	0	1.2
Processor 3	∞	1.2	0

Table 4.5. Distance Cost Matrix

---

[1] Note that we will be using interchangeably the term IMC and IPC.

2.   The Approach

The first attempt was to separate tasks according to their data dependency, which was determined by calculating the several slices of the prototype. Informally, a slice is defined as the set of possible paths from a sink node (nodes with no output) to a root node (nodes with no input edges), i.e., a slice contains all ancestors of a sink node. For a formal definition see Dampier [Dam94]. Clearly, an operator can belong to more than one slice.
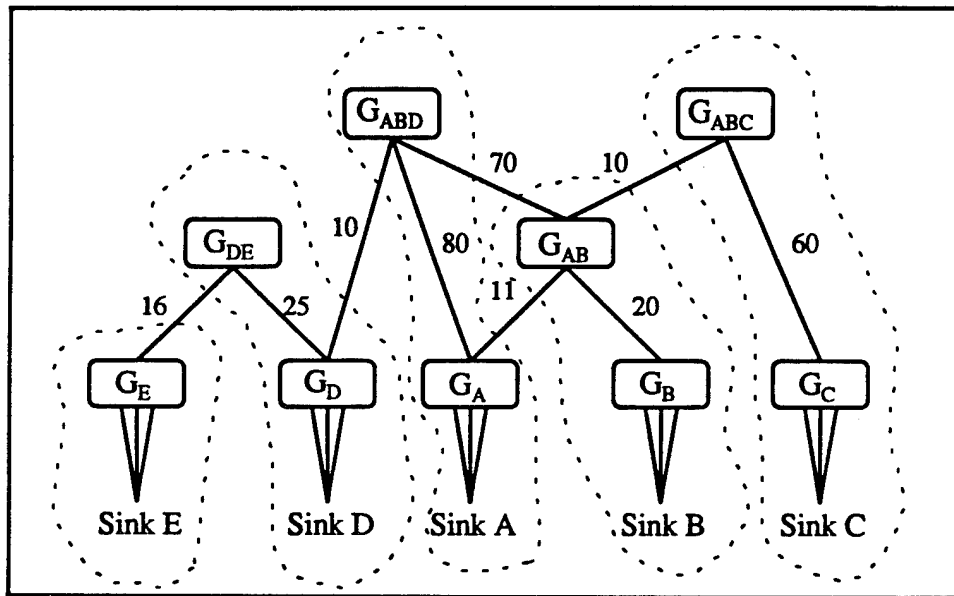
Figure 4.14. The Data Dependency Graph

After all slices are calculated the operators that belong to the same slices are grouped into equivalent classes, such as G_A, G_{AB}, G_{CDE} etc., meaning that they belong to slice A, slices A and B, or slices C, D, and E, respectively. The resulting graph is the Data Dependency Graph, which is shown in Figure 4.14. The following algorithm can then be applied:

1) Pick those operators that belong to two slices. At least one operator must exist in this equivalence class that has two edges, one for each of the slices it belongs to. Pick the least expensive edge, i.e., the one with the least IMC cost, and add the operator to this group. This may later prove to be something less than the

96

best choice, but for now it is the best option available without resorting to the expensive method of checking the entire slice. The final partition is illustrated by the dotted line in Figure 4.14, and presents a cost of 117 IMC units. To get rid of this problem, instead of trying to join in a bottom-up fashion, the most expensive edge not yet included in any group may be added, and an attempt can be made to unite both groups, resulting in the following partition: $\{G_A, G_{ABD}, G_{AB}\}$, $\{G_{ABC}, G_C\}$, $\{G_{DE}, G_D\}$, $\{G_B\}$ and $\{G_E\}$, which has a cost of 56 IMC units;

2) Keep doing this for the operators belonging to three slices, four, etc., until all operators have been processed.

3) If the load factor in some set exceeds one or some specified threshold then the set should be split into two by recursively applying the two-dimensional minimal-cut algorithm, until all sets have a load factor less than one. Note that since the min-cut algorithm is trying to minimize the cost of the edge, it may well not be an optimal choice for minimizing load factor. Checking for load factor is left until the end because the relative costs of those edges could not be determined prior to completing the first two steps..

The intended result was to have several fairly data independent sub-graphs that could be assigned to different processors, having a minimum IPC cost, and, most importantly, providing a very nice modularization for the system with direct effects on reliability. For example, if some processor had a problem, only those modules allocated in that processor would fail. Of course, this approach did not take into account load balancing, but at least provided a starting point.

Unfortunately, after running a partial implementation of this algorithm with several random generated prototypes, its computation cost proved to be very high and most of the prototypes ended up having very few slices to start with.

97

After analyzing the advantages and disadvantages of the initial attempt and several other alternatives, it was decided to use the inter-module communication cost (IMC) as the main cost function, without taking into consideration any data dependency.

Now it is necessary to come up with a consistent way of assigning the IMC cost to each pair of operators in a PSDL graph.

Clearly, in the PSDL context, where complex ADTs can travel through the streams, the amount of data transferred by a stream is variable, and its actual size can only be known at run-time when the actual prototype is executing. Therefore it is necessary to use some kind of average or normalized value, so that the deviations are diminished. Another assumption to be made (it is actually already part of the PSDL model) is that every operator, when fired, outputs one and only one value per firing for each of its output streams. Furthermore, the worst case is assumed, where, once activated, the operator will always produce an output, even if the data triggering conditions or the output guards are not satisfied.

The IMC cost, represented as IMC_INDEX, and the actual amount of data to be transmitted between two operators, denoted as IMC_PER_SEC, are calculated according to the algorithm described in Figure 4.15.

```
for each pair of operators loop
    if parent operator is TC then
        IMC_PER_SEC := CONNECTIVITY × AVG_PROC_TIME × 1000 / PERIOD_PRODUCER;
    elsif parent is NTC then
        IMC_PER_SEC := CONNECTIVITY × AVG_PROC_TIME × 1000 / HARMONIC_BLOCK;
    end if;
    IMC_INDEX := IMC_PER_SEC / NORMALIZED_LOAD_FACTOR
end loop;
```

Figure 4.15. Algorithm for Calculating the IMC Cost Function

Note that in order to quantify and compare IMCs it was necessary to fix the time window for measurement and the *second* was chosen.

AVG_PROC_TIME is the estimated average time in microseconds taken for that system to output a typical PSDL stream to some buffer, which will be later transmitted to

the network. Note that this parameter is innocuous, since it is a constant for every stream. The only reason to maintain the parameter is to make the resulting index more realistic.

CONNECTIVITY is defined as the number of streams connecting two operators including the state streams.

The ratio 1000 ms/ PERIOD (ms) for the time-critical operator specifies the number of periods that occurs in one second, that is, the number of times the producer will fire. For the non-time-critical operator the HARMONIC BLOCK (HB) is used as if there was only one occurrence of the NTC operator in each HB.

Finally, for the IMC_INDEX the NORMALIZED_LOAD_FACTOR is introduced, defined as:

$$(\text{LOAD\_FACTOR PARENT} + \text{LOAD\_FACTOR CHILD}) / \text{MAX\_LF\_PER\_PROC}$$

Note that the above formula is valid for any case except when both operators are NTCs. In this case the formula is changed to:

$$((1.0 - \text{MAX\_LF\_PER\_PROC}) + (1.0 - \text{MAX\_LF\_PER\_PROC})) / \text{MAX\_LF\_PER\_PROC}$$

or

$$(2.0 / \text{MAX\_LF\_PER\_PROC}) - 2.0$$

The rational behind these formulas is that if there are two small LF operators connected by a stream with some IMC_PER_SEC, the IMC_INDEX or, rather, the relative cost for placing them in different processors should be much higher than if they had big load factors, for a same IMC_PER_SEC value. For streams connecting two NTC operators that don't have an explicit load factor, since they don't have periods nor METs, the remaining load factor will be used. In other words, 1.0 - TOTAL_LF, as if it was the load factor. If the load factor is bigger than one, then there must be more than one processor, so that the maximum average load factor per processor is used instead, assuming that the minimum number of processors is available.

Although it is not used in the current implementation, it seems to be a good idea to divide the remaining LF among all NTCs operators. This way it would be less costly to split two NTCs, where the total load factor of the prototype is 0.8, than to split two TC

operators both with load factors 0.2. In the current implementation, both cases have the same cost.

3.   The Current Implementation

As the very first step, the allocation algorithm builds a priority queue of edges in decreasing order of inter-module communication cost (IMC_INDEX), which were previously calculated. Note that it will contain all edges in the prototype and not only those connecting time-critical operators.

Once the priority queue exists, each operator is allowed to form a set by itself. Next a union-find algorithm is applied, so that if the origin and destination operators of the edge being examined belong to different sets, they are united (as long as their combined load factor is still under some threshold previously established by the user).

```
begin -- allocate
    -- Build a priority queue of edges in decreasing order of IMC_INDEX
    BUILD_PRI_QUEUE(COUNT);
    -- Let each operator form a distinct set by itself.
    for I in 1..NEW_GRAPH_PKG.ARRAY_SIZE loop
        OP := NEW_GRAPH_PKG.RETURN_OP(I);
        OP_UNION_FIND_PKG.CREATE(OP_LINK(I),OP);
    end loop;
    while IMC_PRIORITY_QUEUE.NON_EMPTY(PRI_QUEUE) loop
        EDGE := IMC_PRIORITY_QUEUE.READ_BEST(PRI_QUEUE);
        ROOT_A := OP_UNION_FIND_PKG.FIND(OP_LINK (EDGE.ORIGIN));
        ROOT_B := OP_UNION_FIND_PKG.FIND(OP_LINK (EDGE.DEST));
        if not OP_UNION_FIND_PKG.eq (ROOT_A, ROOT_B) then
            if ROOT_A.LF + ROOT_B.LF ≤ ALLOCATION_FACTOR then
                ROOT_C := OP_UNION_FIND_PKG.UNION(ROOT_A, ROOT_B,
                                                  ALLOCATION_FACTOR);
            end if;
        end if;
        IMC_PRIORITY_QUEUE.REMOVE_BEST(PRI_QUEUE);
    end loop;
end allocate;
```

Figure 4.16. Partial View of the Allocation Program

As can be seen, the current approach is a kind of first-fit bin-packing, where the size of the bin is dictated by the ALLOCATION FACTOR specified by the user. A very

simple modification which would allow a better load balancing is to substitute the ALLOCATION FACTOR by the AVERAGE PROCESSOR LOAD FACTOR of the prototype, multiplied by some number, for example, 1.1, to allow some variation around the average. In doing this, it is being enforced that all processors will get an even load, despite of an increase in the communication cost. Other checks could be applied as well, such as checking the requirements or the placement cost matrix to see if the operators could be allocated to the same processor, or if they needed to be in a specific processor. The slices they belong to could also be examined, so that even if the load balancing rule is not completely satisfied they could still be assigned to the same processor if they were in the same slice. As can be seen, there are an enormous number of possibilities for cost functions. However, finding the one that best fits the application requires a great deal of fine tuning.

The union-find data structure has been implemented as an in-tree, where the nodes can have many children, therefore, after all the sets have been formed, we need an $O(n^2)$ worst case algorithm in order to retrieve their members. Another way to implement it that would make the retrieve operation much cheaper is by using a double linked list, but then the insert operation would be a little bit more expensive. In both cases, the union-find algorithm could be enhanced by adding path compression and balancing into the implementation, resulting in an $O(m \log n)$ time algorithm, where m is the number of edges in the graph.

Finally, the allocation algorithm outputs a set of sets, i.e., a set where each of the components is another set containing the nodes in that partition. Although not included in the current implementation, it should ultimately output a map instead of a set, where each of the partitions would be mapped to a specific processor, according to the requirements.

V. ARCHITECTURAL ISSUES OF THE CAPS SCHEDULER

Section A of this chapter describes several issues related to the architecture of the CAPS scheduler in its current uniprocessor implementation. Section B presents a novel architecture for dealing with the distributed scheduling case. The remaining sections of this chapter contain a proposed implementation, first using the current available technology and then using the upcoming facilities offered by Ada95. It is important to note, however, that while implementing the distributed system in Ada provides a uniform environment for building prototypes, it suffers from the disadvantage that tasking and the new distributed systems support in Ada95 are not time-bounded. Hence, in order for the distributed Ada prototype to satisfy the timing constraints as specified, the average behavior of the underlying host operating system and the network communication sub-system must be relied upon.

A. THE CURRENT SCHEDULER - UNIPROCESSOR ARCHITECTURE

Currently, CAPS is a development environment, implemented in the form of a collection of tools, that are linked together by a user interface. The prototyping process is accomplished by running several tools independently, one after the other, so that their output taken together make up the final Ada program, which will implement the supervisory control of the prototype.

More specifically, the translator converts the PSDL program defined by the user into compilable Ada units. During this process, it creates the following five major packages: exceptions, instantiations, timers, streams, and drivers, all preceded by the name of the prototype followed by an underscore. Ultimately each of these will become part of the prototype supervisory Ada program.

The first three of these packages contain all of the user declared exceptions, generic packages and timer instantiations defined in the PSDL program. The package streams contains the instantiations of all the streams used by the prototype, which are implemented as Ada generic tasks contained in the generic package PSDL_STREAMS,

103

which contains all stream types supported by PSDL. A partial view of the supervisory program for the Patriot Missile prototype is shown in Figure 5.1.

```
package PATRIOT_EXCEPTIONS is
   -- PSDL exception type declaration
   type PSDL_EXCEPTION is (UNDECLARED_ADA_EXCEPTION);
end PATRIOT_EXCEPTIONS;

package PATRIOT_INSTANTIATIONS is
   -- Ada Generic package instantiations
end PATRIOT_INSTANTIATIONS;

   with PSDL_TIMERS;
package PATRIOT_TIMERS is
   -- Timer instantiations
end PATRIOT_TIMERS;


   -- with/use clauses for atomic type packages
   -- with/use clauses for generated packages.
   with PATRIOT_EXCEPTIONS; use PATRIOT_EXCEPTIONS;
   with PATRIOT_INSTANTIATIONS; use PATRIOT_INSTANTIATIONS;
   -- with/use clauses for CAPS library packages.
   with PSDL_STREAMS; use PSDL_STREAMS;
package PATRIOT_STREAMS is
   -- Local stream instantiations
   package DS_INTERCEPT_ANGLE_CONTROL_PATRIOT is new
   PSDL_STREAMS.FIFO_BUFFER(FLOAT);
   package DS_LAUNCH_ANGLE_LAUNCH_PATRIOT is new
   PSDL_STREAMS.FIFO_BUFFER(FLOAT);
   package DS_LAUNCH_STATUS_SCUD_RADAR is new
   PSDL_STREAMS.SAMPLED_BUFFER(LAUNCH_STATUS_RECORD);
   package DS_LAUNCH_STATUS_DISPLAY_SCUD is new
   PSDL_STREAMS.SAMPLED_BUFFER(LAUNCH_STATUS_RECORD);
   package DS_LAUNCHER_POSITION_SCUD_RADAR is new
   PSDL_STREAMS.SAMPLED_BUFFER(FLOAT);
   package DS_MISSILE_TRACK_CHECK_THREAT is new
   PSDL_STREAMS.SAMPLED_BUFFER(TRACK);
   package DS_SCUD_STATUS_DISPLAY_SCUD is new
   PSDL_STREAMS.SAMPLED_BUFFER(MISSILE_STATUS);
   package DS_SCUD_TRACK_DISPLAY_SCUD is new
   PSDL_STREAMS.SAMPLED_BUFFER(TRACK);
   package DS_TACTICAL_STATUS_DISPLAY_TACTICAL is new
    PSDL_STREAMS.SAMPLED_BUFFER(MISSILE_STATUS_RECORD);
   package DS_TARGET_RANGE_CONTROL_PATRIOT is new
   PSDL_STREAMS.FIFO_BUFFER(FLOAT);
   -- State stream instantiations
end PATRIOT_STREAMS;
```

Figure 5.1. Partial View of Patriot.a

104

Currently, CAPS implementation supports only the sampled streams where data can always be written and read, the state streams, which are basically a sampled stream with an initial value, and the data flow streams, which are implemented as a FIFO buffer with size one. The streams are implemented as individual Ada tasks with entries such as READ, WRITE and CHECK, whose implementation will vary according to the type of stream.

Finally, the package drivers basically contains all of the data declarations, the data trigger checks that control whether a stream should or should not be read, the execution trigger checks that decide whether or not to fire the operator, and the output guard checks, which will allow whether or not an output is to be written to the output streams. Each of these checks are implemented in the following way:

1. Data Triggers

If an operator has no triggering condition at all, its input streams will be read whenever the operator is fired, but they will never generate any overflow or underflow exceptions. Similar situation happens when the streams are state streams.

If at least one of the incoming streams is a TRIGGERED BY SOME sampled stream, then the streams will be read whenever one or more of the streams in the TRIGGERED BY SOME set has new data, but again, they will never generate an underflow exception. Because of this, care must be taken with respect to the very first reading of data from sampled streams, since garbage may be consumed.
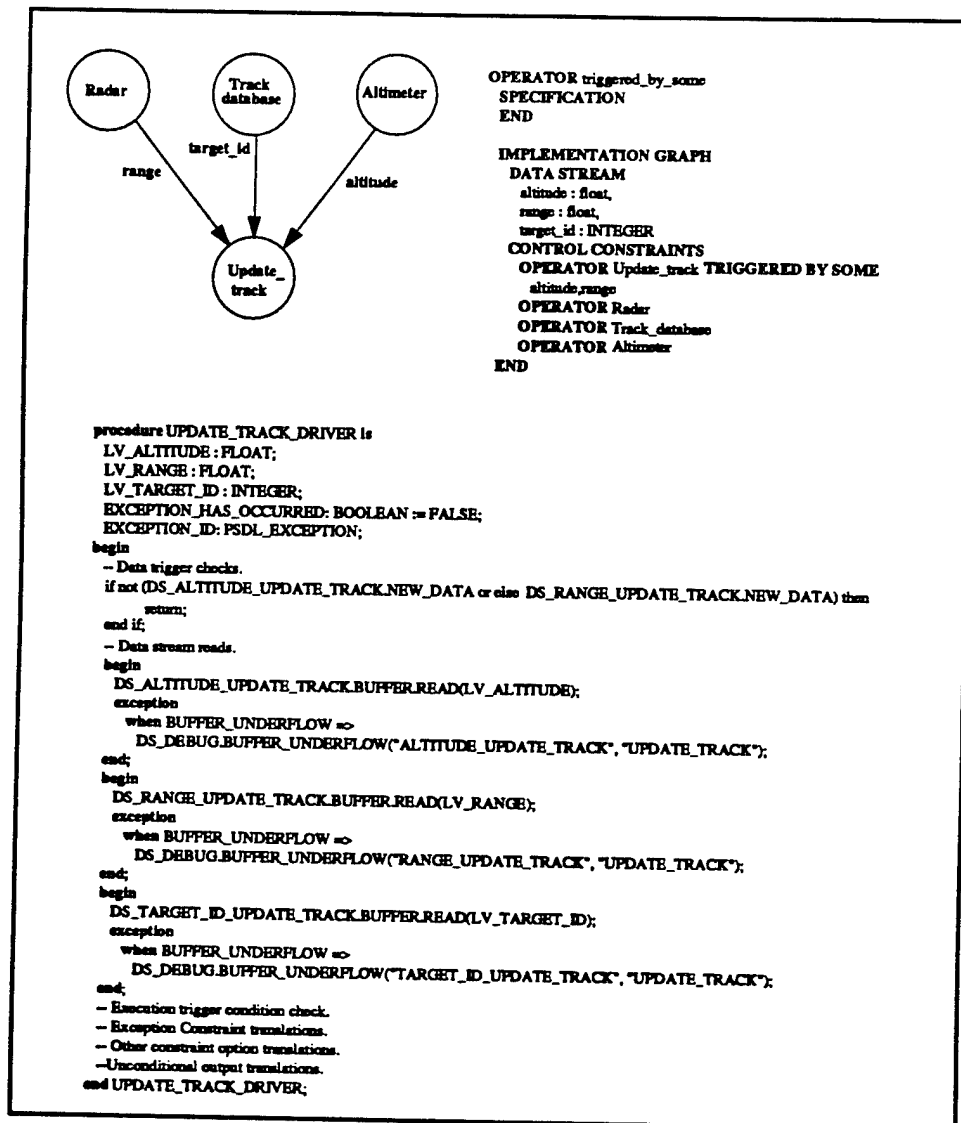
Figure 5.2. TRIGGERED BY SOME Implementation

If at least one of the incoming streams is a data flow stream, in other words, has a TRIGGERED BY ALL condition, the streams will only be read if the data flow stream has a new value in its buffer, and any attempt to read an old value from a data flow stream, will generate an underflow exception. As shown in Figures 5.2 and 5.3, the read operation is actually a call to rendezvous with the READ entry of the incoming stream task.
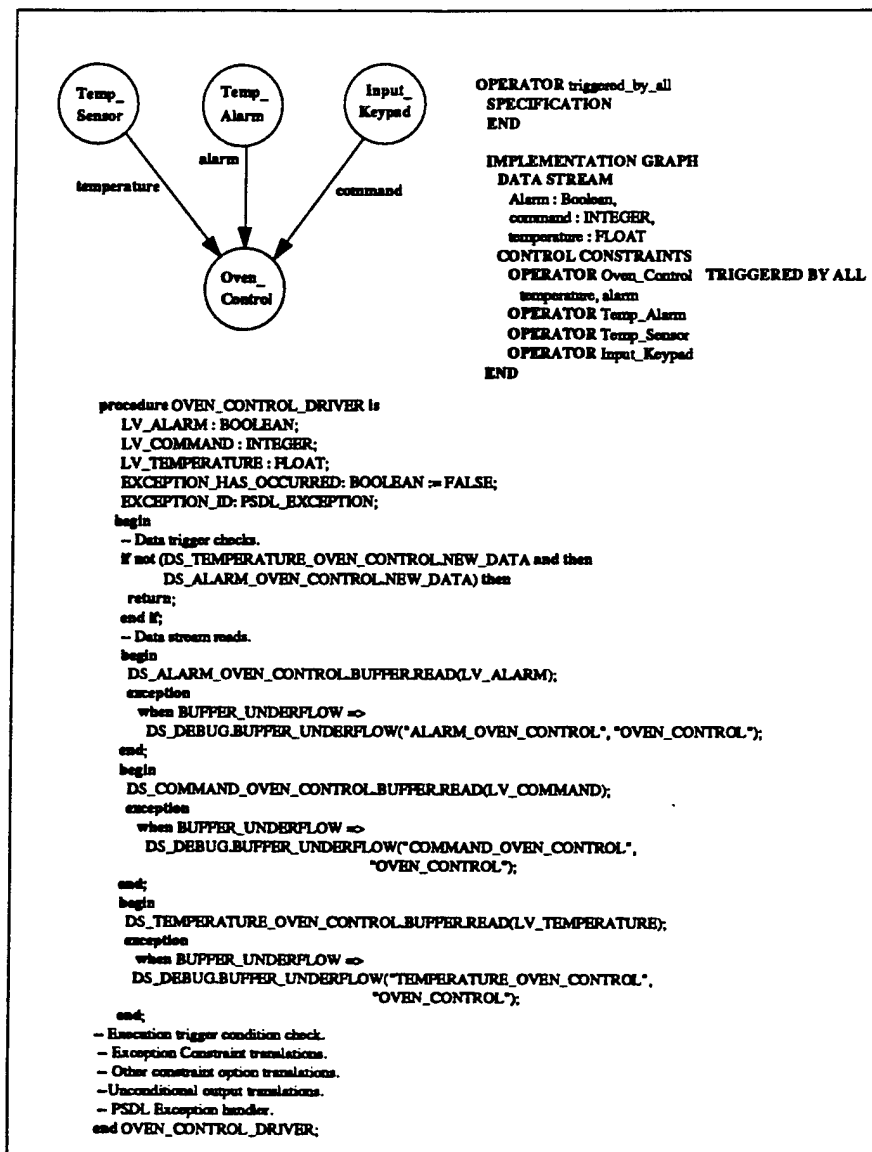
Figure 5.3. TRIGGERED BY ALL Implementation

2. Execution Triggers

The execution trigger is where the actual program that implements the functionality of that operator, which is provided by the user, will be called if the conditions are satisfied. These conditions come from the TRIGGERING IF part of the PSDL program. Note that even if they are not satisfied, the data has already been consumed, and is therefore marked as old data.
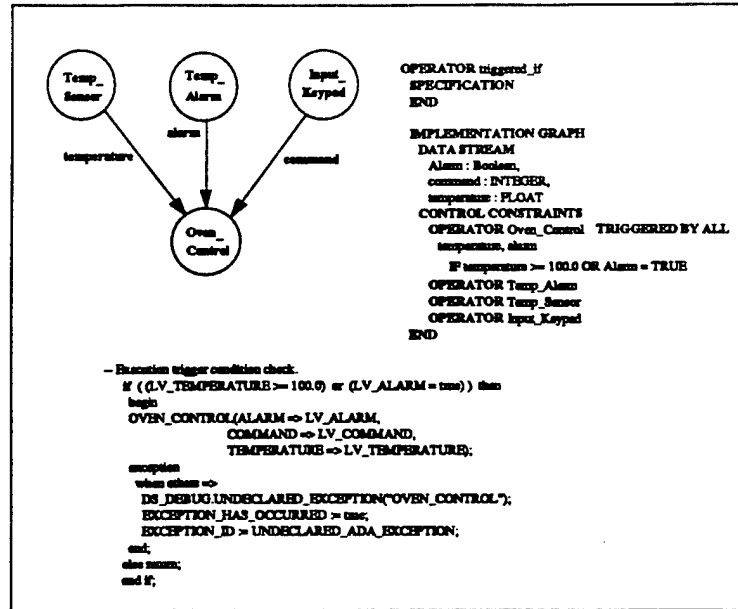
Figure 5.4. TRIGGERING IF Implementation

3. Output Guards

Finally, the output guards are checked. If the conditions are satisfied, a rendezvous with the output stream tasks is requested by calling their WRITE entry.
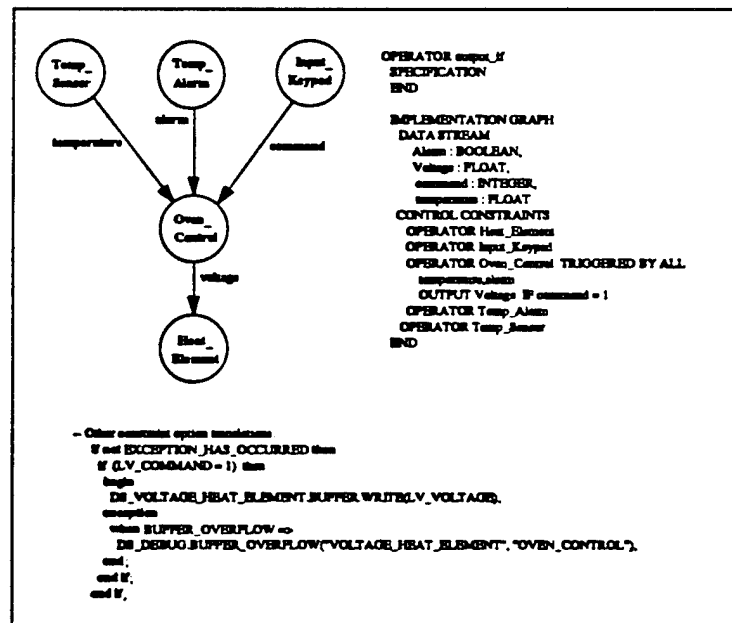
Figure 5.5. Output Guards Implementation

Besides these packages that are generated by the Translator, there are another two packages generated by the Static Scheduler and by the Dynamic Scheduler. When consolidated by one of the CAPS scripts, they will form the so called *prototype supervisory program*, receiving the name of the prototype followed by a ".a" extension, which stands for an Ada program.
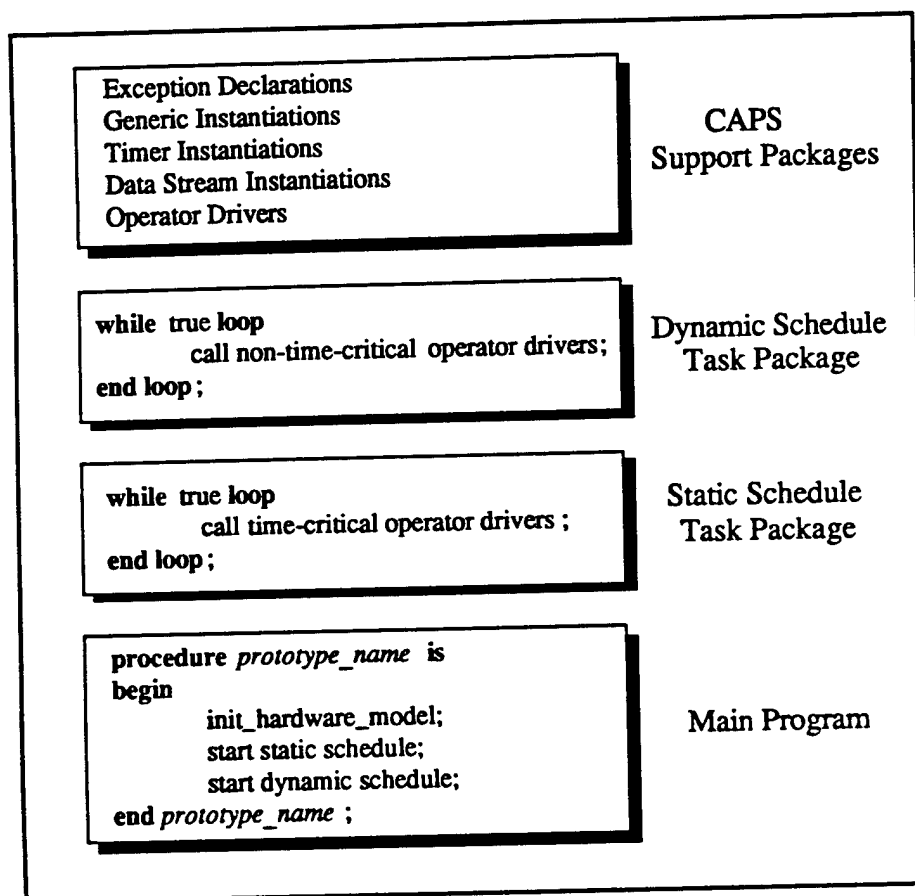
Figure 5.6. CAPS Supervisory Program Structure

CAPS is composed of four major Ada tasks with the following priorities, as defined in the package PRIORITY_DEFINITIONS:

1) *Debugger Task* - it handles all CAPS debugging tools used during prototype execution, and has the highest priority within CAPS, which is 4

2) *Stream Tasks* - each stream is implemented as one Ada task with priority 3

3) *Static Scheduler Task* - it is responsible for calling all the timing critical operators, according to the static schedule. The TC operators will be called in a non-preemptive way, so that each instance of an operator will execute to completion; being preempted only by the debugger task, or during operations with the stream tasks. It has a priority of 2. Note that, although the stream tasks have higher priority, they are called (synchronized) by this task, so that there will be no problems such as another stream from another operator trying to gain control of the CPU.

4) *Dynamic Scheduler Task* - it is assigned the lowest priority (priority 1) within CAPS, and it handles all the non-time critical operators of the prototype. They will run in a pre-defined order established by the dynamic schedule, whenever there is idle time in the static schedule. The NTC operators, due to their low priority, can be preempted by any other task and, as a matter of fact, they are not even guaranteed to run at all. This problem of unbounded blocking will be addressed later on.

B. THE PROPOSED DISTRIBUTED ARCHITECTURE

In the uniprocessor case, the translator had no information about the output of the scheduler. For the distributed case, however, this information is crucial, since it will have to generate different Ada units for each of the processors involved in the prototyping.

Once the scheduler has generated the different partitions, defining which operator belongs to which partition, the translator will have to be called, so that it can generate as many supervisory files as the number of partitions. It is suggested that the prototype name followed by the partition number be used as the naming convention for the supervisory files, e.g., PATRIOT_1.a, PATRIOT_2.a, and so on.

The following information should be passed by the scheduler to the translator, so that it can perform its job:

1) Number of partitions and a list with the operator names belonging to each partition

2) Mapping from partitions to processors according to the requirements

For the sake of simplicity, it is assumed that there is a homogenous cluster of processors, so that a configuration of partitions is not needed. The process of mapping the partitions of a program to the nodes in a distributed system is called configuring the partitions. Note, however, that even after having abolished condition 2, there is still a need to provide the translator with the name of the processors. It is suggested that this information come from the CAPS interface.

Once this information is available to the translator, it should generate a supervisory file for each partition, exactly as it did for the uniprocessor case, except for the following differences:

1) In the new package streams, where the streams are instantiated, if a specific stream is going to some operator external to that partition, and only in that case, it should be hard-coded as an instantiation of a special and newly created kind of stream, i.e., the network stream. Note that this stream has only one entry, which is *write_external*, considering that all reads will be to local streams. Certainly, the package PSDL_STREAMS will have to be totally changed to conform with the new model for distributed scheduling without synchronization, which requires a buffer size of three for the network streams. Another modification made in this package relates to the sampled streams, which are now divided into two groups, non-triggering (NT) and TRIGGERED BY SOME (TBS), since they have quite different semantic behaviors. Figure 5.7 shows the specification of the new package containing the stream tasks.

```
with PRIORITY_DEFINITIONS;                      -- Implements a buffer with size 3, for states streams
use PRIORITY_DEFINITIONS;                       -- that have triggering "BY SOME" condition (TBS)
package PSDL_STREAMS is                          generic
    BUFFER_OVERFLOW : exception;                 type ELEMENT_TYPE is private;
    BUFFER_UNDERFLOW: exception;                 INITIAL_VALUE: ELEMENT_TYPE;
                                                 package TBS_STATE_BUFFER is
                                                     task BUFFER is
    -- Implements a buffer with size 1, for sampled   pragma PRIORITY(BUFFER_PRIORITY);
    -- streams with no triggering condition (NT)       entry CHECK(NEW_DATA: out BOOLEAN);
    generic                                            entry READ(VALUE: out ELEMENT_TYPE);
    type ELEMENT_TYPE is private;                      entry WRITE(VALUE: inELEMENT_TYPE);
    package NT_SAMPLED_BUFFER is                     end BUFFER;
        task BUFFER is                              function NEW_DATA return BOOLEAN;
            pragma PRIORITY(BUFFER_PRIORITY);   end TBS_STATE_BUFFER;
            entry READ(VALUE: out ELEMENT_TYPE);
            entry WRITE(VALUE: in ELEMENT_TYPE);
        end BUFFER;                              -- Implements a buffer with size 3, for dataflow
    end NT_SAMPLED_BUFFER;                       -- streams, that is, those that have the triggering
                                                 -- "BY ALL" condition
                                                 generic
    -- Implements a buffer with size 3, for sampled type ELEMENT_TYPE is private;
    -- streams that have triggering "BY SOME"     package FIFO_BUFFER is
    -- condition (TBS)                                task BUFFER is
    generic                                            pragma PRIORITY(BUFFER_PRIORITY);
    type ELEMENT_TYPE is private;                      entry CHECK(NEW_DATA: out BOOLEAN);
    package TBS_SAMPLED_BUFFER is                       entry WRITE(VALUE: in ELEMENT_TYPE);
        task BUFFER is                                  entry READ(VALUE: out ELEMENT_TYPE);
            pragma PRIORITY(BUFFER_PRIORITY);       end BUFFER;
            entry CHECK(NEW_DATA: out BOOLEAN);     function NEW_DATA return BOOLEAN;
            entry READ(VALUE: out ELEMENT_TYPE); end FIFO_BUFFER;
            entry WRITE(VALUE: in ELEMENT_TYPE);
        end BUFFER;
        function NEW_DATA return BOOLEAN;        -- Implements a buffer with size 1, for networked
    end TBS_SAMPLED_BUFFER;                       -- stream, no matter what kind of streams they are
                                                 with A_STRINGS;  use A_STRINGS;
    -- Implements a buffer with size 1, for state streams with ADA_STREAMS;
    -- that have no triggering condition (NT)     with SYSTEM_RPC;
    generic                                      generic
    type ELEMENT_TYPE is private;                type ELEMENT_TYPE is private;
    INITIAL_VALUE: ELEMENT_TYPE;                 PROC : SYSTEM_RPC.PARTITION_ID;
    package NT_STATE_BUFFER is                   STREAM_NAME : in A_STRING;
        task BUFFER is                           package NETWORK_BUFFER is
            pragma PRIORITY(BUFFER_PRIORITY);        task BUFFER is
            entry READ(VALUE: out ELEMENT_TYPE);     pragma PRIORITY(BUFFER_PRIORITY);
            entry WRITE(VALUE: inELEMENT_TYPE);      entry WRITE_EXTERNAL(
        end BUFFER;                                      VALUE: in ELEMENT_TYPE;
    end NT_STATE_BUFFER;                                 PROC : in SYSTEM_RPC.PARTITION_ID;
                                                         STREAM_NAME : in A_STRING);
                                                     end BUFFER;
                                                 end NETWORK_BUFFER;
                                             end PSDL_STREAMS;
```

Figure 5.7. The New PSDL_Streams Ada Package Specification

2) The new drivers package should contain only the driver procedures related to the operators belonging to that partition. It is very important to notice that the distributed scheduling model assumes that a stream resides, i.e., it is

instantiated, in the same processor or partition of its consumer operator.[1] Therefore, for the consumer operator, it is irrelevant where the data came from, and, furthermore, no changes will be needed for the individual driver procedures within this package, since all the reads will be to local streams. The only change would occur if it was necessary to perform a write to an external operator. In this case, the write operation should be hard-coded by the translator as a call to *write_external*, an entry of the special network stream task. In Figure 5.8, which presents the network stream task body, it is apparent that, after this rendezvous is accepted, there should be a call to some inter-processor communication routine, e.g., DO_APC, that would deliver the message. It is also at this point where most of the problems are going to appear, as shall be seen.

```
with A_STRINGS; use A_STRINGS
with ADA_STREAMS;
with SYSTEM_RPC;
package body NETWORK_BUFFER is
    task body BUFFER is
        PARAMETERS : SYSTEM_RPC.PARAMS_STREAM_TYPE(3);
        -- This type allows multiple stream elements within the
        -- same stream, depending on its declaration
    begin
        loop
            accept WRITE_EXTERNAL(VALUE: in ELEMENT_TYPE;
                        PROCESSOR : in SYSTEM_RPC.PARTITION_ID;
                        STREAM_NAME : in A_STRING) do
                SYSTEM_RPC.DO_APC(PROCESSOR,PARAMETERS);
                -- parameters will include the remote procedure name,
                -- the psdl_stream_name and value
            end WRITE_EXTERNAL;
        end loop;
    end BUFFER;
end NETWORK_BUFFER;
```

Figure 5.8. Body of the Network Stream Task

---

[1] This assumption will require that all exceptions from external streams should be treated and consequently hard-coded in the consumer's side.

The changes made so far are very minor, since most of the burden is being put on the write operation to external streams. In fact, the most difficult part of this implementation is finding a way to receive the incoming messages from the different processors and operators. Some kind of communications server, that will have the duty of receiving and routing all the incoming messages to its final destination, will be needed. Due to the semantics of PSDL, in order to reliably implement this communication, it will be necessary to send some kind of header containing the consumer operator, the name of the stream and the name of the destination processor along with the data.

These requirements for the header come from situations such as when the same operator is trying to write to the same stream into different operators in different partitions. This case is illustrated in Figure 5.9. In the next section the different options available for implementing this communication sub-system are described.
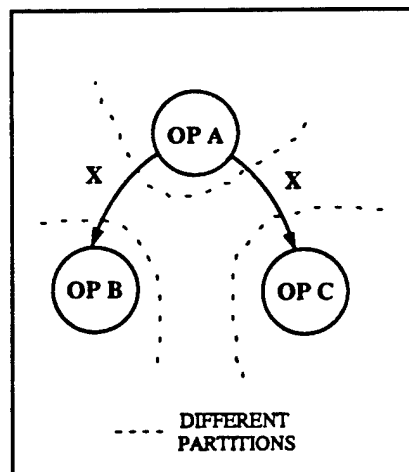
Figure 5.9. Justification for the Header Information

C. IMPLEMENTATION ISSUES OF THE COMMUNICATION SUBSYSTEM

One of the most important design issues is the choice of the communication subsystem. It is recommended to use the remote procedure call (RPC) paradigm as opposed to the traditional message passing mechanism. The reasons for this choice is that RPC is widely implemented for interprocess communication between computers across a network, being supported by most of the emerging distributed operating systems. Several

114

standards have been initiated by organizations, such as ISO and OSF. This method also provides an asynchronous form, relaxing the original synchronous semantics of RPC. Finally, the Annex E (Distributed Systems) of the Ada95 Reference Manual makes it the choice, though not mandatory, for future implementations of this Annex.[Ada95]

1.   The RPC Model

The remote procedure call model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well-specified location. It then transfers control to the procedure, and eventually gains back control. At that point, the results of the procedure are extracted from the well-specified location and the caller continues execution.[Sun90]

The remote procedure call is similar. That is, the caller process sends a call message to the server process and waits (blocks) for a reply message. The call message contains the procedure's parameters, among other things. The reply message contains the procedure's results, among other things. Once the reply message is received, the results of the procedure are extracted, and the caller's execution is resumed.[Sun90]

Note that in this model, only one of the two processes is active at any given time. The RPC protocol, however, makes no restriction if the implementation allows the calling routine to do some useful work while waiting for the reply (asynchronous mode).

2.   The First Approach

The first idea was to implement the RPC paradigm by using the standard RPC libraries. However, in order to do that within CAPS, it would be necessary to call from inside an Ada task, more specifically from inside the network tasks, a C routine that would implement the RPC calls (see Figure 5.8). The reason for a C routine is that there is no library support or existing bindings for implementing RPC from inside Ada83. It would not be difficult to write an Ada wrapper to the C routine. However, the biggest problem to be dealt with is how to pass the Ada parameters to the C routine, which could be very complicated abstract data types from the PSDL prototype. Assuming that this problem

115

could somehow be solved, there is an additional problem: How could this C routine pass the complex ADTs through the streams? In the Unix/C world, there currently exists a great deal of support for these kinds of operations.

For example, the *rcpgen* utility is basically a compiler that accepts a remote program interface definition written in the RPC language, which is very similar to C, and outputs a C program, containing all the client routines, the server routine, and most importantly, all the XDR filter routines. An XDR routine converts procedure arguments and results in the network format (sequential streams) and vice-versa.

The External Data Representation (XDR) standard comprises a set of library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. XDR is the backbone of Sun's RPC package, in the sense that data for remote procedure calls is transmitted using this standard. It was designed to work across different languages, operating systems, and machine architectures.

It is important to note, however, that most of the time required to prepare a data structure for transfer is not spent in conversion but in traversing the elements of the data structure. To transmit a tree, for example, each leaf must be visited and each element in a leaf record must be copied to a buffer and aligned there. Storage for the leaf may have to be deallocated after the data is sent. Similarly, to receive a tree, storage must be allocated for each leaf, data must be moved from the buffer to the leaf and properly aligned, and pointers must be constructed to link the leaves together. [Sun90]

In this case what is needed is a remote procedure called *receive*, running in all the machines, ready to intercept any incoming messages, and another routine, namely *send*, that will also run in all machines and will remotely call the *receive* routine. In Figure 5.10 both routines which were successfully tested in the "C" environment are presented. Note that the *send* routine is not sending anything, but merely passing parameters to the remote procedure *receive*.

116

```
RPC_REC.C
/* receiver.c - remote procedures; called by server
stub. */

#include <stdio.h>
/* standard RPC include file */
#include <rpc/rpc.h>
/* this file is generated by rpcgen */
#include "RPC_receive.h"

/* Receive a string of chars and reply with a status
*/

char **
receive_1(message)
   char ** message;
{
   static char status[20] = "OK";
   static char ptr[100];
   static char *ptr1;

   printf("Received message = %s\n", *message);
   fflush(stdout);
   ptr1 = &status[0];
   strcpy (ptr,*message);
   ptr1 = &ptr[0]; */
   return(&ptr1);
}


RPC_SEND.C
/* RPC_send.c - client program for remote receive
service.*/

#include <string.h>
#include <stdio.h>
/* standard RPC include file */
#include <rpc/rpc.h>
/* this file is generated by rpcgen */
#include "RPC_receive.h"
```

```
main(argc, argv)
   int argc;
   char *argv[];
{
   CLIENT    *cl;    /* RPC handle */
   char    *receiver_name;
   char    **status;
   char    *message;

   if (argc != 3) {
      fprintf(stderr, "usage: %s hostname
                       message\n", argv[0]);
      exit(1);
   }
   receiver_name = argv[1];
   message = argv[2];
   /* Create the client "handle" */
   if ( (cl = clnt_create(receiver_name,
         DISTR_SCHEDULE, CAPS95, "udp"))
         == NULL) {
   /* Can't establish connection with receiver */
      clnt_pcreateerror(receiver_name);
      exit(2);
   }


   /* call the remote procedure "receive_1" */
   printf("Message to be transmited = %s\n",
         message);
   fflush(stdout);
   if ((status = receive_1(&message, cl)) ==
      NULL) {
      clnt_perror(cl, receiver_name);
      exit(3);
   }
   printf("Status from remote receiver %s is
                  %s\n", receiver_name,*status);
   clnt_destroy(cl);    /* done with the handle */
   exit(0);
}
```

Figure 5.10. The RPC Programs for the New Scheduler

Finally, if both problems have been solved, i.e., the parameter passing between C
and Ada in the sender side and the Ada bindings for the XDR routines, there is still an

117

additional problem in the receiver side due to the way RPC is now implemented in C. The receiving, or the server, routine, is implemented as a forever loop by calling the Unix system call *svc_run()*. To overcome this problem one would need to be able to call an Ada procedure from inside a C routine, and again the same problem of passing parameters would be present.

Another approach, such as using files to exchange data between C and Ada, could be used, but then other problems, such as file locking, and internal synchronization between C and Ada tasking (so that no data could be overwritten before being consumed) would come into play.

Because of all these problems, it seems that a better solution is needed, and just such a solution is present in the Ada95 implementation, which will be described next.

3.   The Ada95 Approach

Annex E defines facilities for supporting the implementation of distributed systems using multiple partitions working cooperatively as part of a single Ada program. These facilities include pragmas for categorizing library units according to the role they play in the distributed system, such as Shared_Passive, Remote_Types and Remote_Call_Interface, and other mechanisms for supporting communication and access to shared data. [Ada95]

The Partition Communication Subsystem (PCS), as defined in Annex E, provides facilities for supporting communication between the active partitions of a distributed program by using the remote procedure call interface (RPC). The annex also proposes a specification for the RPC interface between active partitions within the PCS, which will be contained in the package System.RPC. Figure 5.11 introduces the proposed specification for the package System.RPC.

```
with Ada.Streams;
package System.RPC is
  type Partition_ID is range 0 .. implementation-defined
  Communication_Error : exception;
  type Params_Stream_Type (Initial_size : Ada.Streams.Stream_Element_Count) is new
                          Ada.Streams.Root_Stream_Type with private;

  procedure Read(Stream : in out Params_Stream_Type;
                 Item : out Ada.Streams.Stream_Element_Array;
                 Last : out Ada.Streams.Stream_Element_Offset);

  procedure Write(Stream : in out Params_Stream_Type;
                  Item : in Ada.Streams.Stream_Element_Array);


  -- Synchronous call
  procedure Do_RPC(Partition : in Partition_ID;
                   Params : access Params_Stream_Type
                   Result : access Params_Stream_Type);


  -- Asynchronous call
  procedure Do_APC(Partition : in Partition_ID;
                   Params : access Params_Stream_Type);


  -- The handler for incoming RPCs
    type RPC_Receiver is acess procedure(Params : access Params_Stream_Type
                                         Result : access Params_Stream_Type);
  procedure Establish RPC_Receiver(Receiver : in RPC_Receiver);

private
  -- not specified by the language
end System.RPC;
```

Figure 5.11. Package System.RPC (Specification)

As noted in Figure 5.11, during the execution of a remote subprogram call, most of the parameters (and later results, if any) are passed using a stream oriented representation which is suitable for transmission between partitions. The annex calls this action marshalling. Unmarshalling is the reverse action of reconstructing the parameters or results from the stream-oriented representation. Note that there is not any defined standard for transformation, but nevertheless the XDR standard seems to be the choice for most of the Ada compiler vendors.

119

The type Partition_ID is used to identify a partition, and Params_Stream_Type is used for identifying the particular remote subprogram that is being called, as well as marshalling and unmarshalling the parameters or result of a remote subprogram call, as part of sending them between partitions. The Read and Write procedures override the corresponding abstract operations for the type Params_Stream_Type.

Both synchronous and asynchronous communication are supported, and are implemented by the procedures Do_RPC and Do_APC, respectively. Both procedures send a message to the active partition identified by the Partition parameter. The first one blocks the calling task until a reply message comes from the called partition, or some error is detected by the PCS, in which case Communication_Error is raised at the point of the call to Do_RPC. Do_APC operates in the same way as Do_RPC, except that it is allowed to return immediately after sending the message.

Finally, the procedure Establish_RPC_Receiver is called only once, immediately after elaborating the library units of an active partition, but prior to invoking the main subprogram, if any. The Receiver parameter designates an implementation-provided procedure called the RPC_Receiver which will handle all RPCs received by the partition. Establish_RPC_Receiver saves a reference to the RPC-receiver. When a message is received at the called partition, the RPC-receiver is called with the Params stream containing the message. When the RPC-receiver returns, the contents of the stream designated by Result is placed in a message and sent back to the calling partition.

The implementation of the RPC-receiver shall be reentrant, thereby allowing concurrent calls on it from the PCS to service concurrent remote subprogram calls into the partition.

a. *The Package Streams*

A Stream is a sequence of elements comprising values from possibly different types, and allowing sequential access to these values. A stream type is a type in the class whose root type is Streams. Root_Stream_Type. [Ada95]

The types in this class represent different kinds of streams. The pre-defined stream-oriented attributes like T'Read and T'Write make dispatching calls on the Read and Write procedures of the Root_Stream_Type.

```
package Ada.Streams is
    pragma Pure(Streams);
    type Root_Stream_Type is abstract tagged limited private;
    type Stream_Element is mod implementation-defined;
    type Stream_Element_Offset is range implementation-defined;
    subtype Stream_Element_Count is
            Stream_Element_Offset range 0 .. Stream_Element_Offset'Last;
    type Stream_Element_Array is
            array(Stream_Element_Offset range <>) of Stream_Element;

    procedure Read(Stream : in out Root_Stream_Type;
                    Item : out Stream_Element_Array;
                    Last : out Stream_Element_Offset) is abstract;

    procedure Write(Stream : in out Root_Stream_Type;
                    Item : in Stream_Element_Array) is abstract;

private
    -- not specified by the language
end Ada.Streams;
```

Figure 5.12. Package Ada.Streams (Specification)

Read operations transfer Item'Length stream elements from the specified stream to fill the array Item. The index of the last stream element transferred is returned in Last. Last is less than Item'Last only if the end of the stream is reached.

The Write operation appends Item to the specified stream. There are also the Read, Write, Output and Input attributes that convert values to a stream of elements and reconstruct values from a stream.

For every subtype S of a type T, some attributes are defined, which denote either a procedure or a function call. Figure 5.13 presents such attributes.

121

```
-- writes the value of Item to Stream
procedure S'Write(Stream : access Ada.Streams. Root_Stream_Type'Class;
                  Item : T);

-- reads the value of Item from Stream
procedure S'Read(Stream : access Ada.Streams. Root_Stream_Type'Class;
                 Item : out T);

-- writes the value of Item to Stream, including any bounds or discriminants
procedure S'Output(Stream : access Ada.Streams. Root_Stream_Type'Class;
                   Item : T);

-- reads and returns the value of Item from Stream, using any bounds or
-- discriminants written by a corresponding S'Output
function S'Input(Stream : access Ada.Streams. Root_Stream_Type'Class;
                 return T);
```

Figure 5.13. Stream Attributes

b.   *Conclusions*

All of the problems that have been discussed in this section have been addressed in the Ada95 implementation. Therefore, in order to implement the distributed scheduling model, it is only necessary to follow the directions introduced in Section B. It is now apparent that the example given in Figure 5.8 had already considered the packages (System_RPC and Ada_Streams) and procedures (DO_APC) to be introduced with Ada95. The only part that is not yet clear, because it is dependent upon implementation, is the marshalling and unmarshaling operations, which will affect the manner in which the Ada stream is constructed from the parameters passed during the rendezvous with the write_external entry of the network stream task.

Figure 5.14 presents a pictorial view of the proposed architecture for the new Distributed CAPS Scheduler.
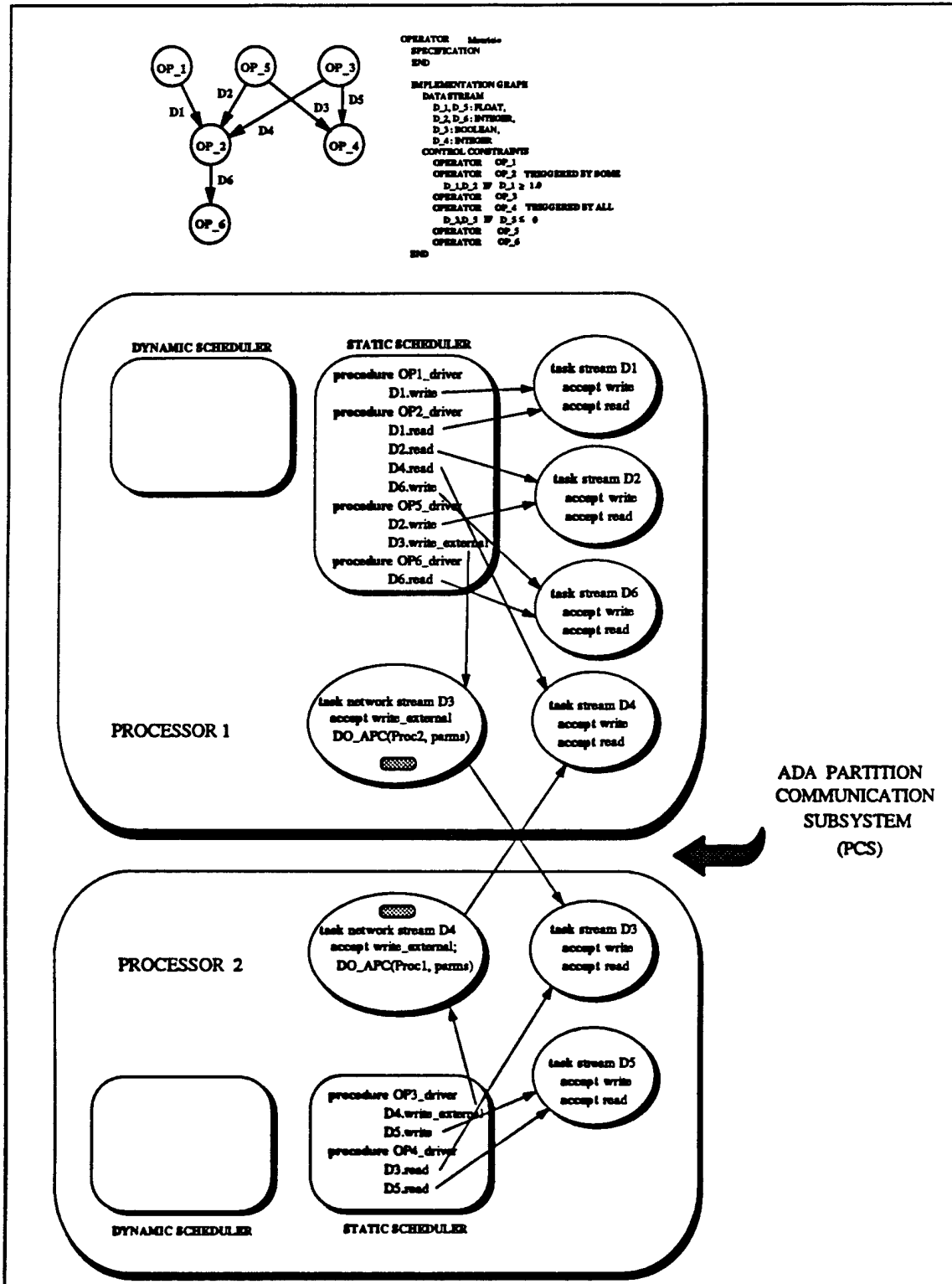
Figure 5.14. Architecture for the Distributed CAPS Scheduler

D. CPU SPEED RATIO ISSUES IN A PROTOTYPING ENVIRONMENT

In a software prototyping environment, where the host machines usually used for prototyping are not similar to the intended target machines (which may not even be known *a priori*), special attention must be taken so that erroneous conclusions due to timing problems during the prototyping are avoided.

There are two kinds of timing errors that can be foreseen in a real-time system. Both of them may cause undesirable system behavior, such as deadlocks, buffer overflows, or data inconsistency. The first kind of error has a relative nature, since it is caused by computational events that occur in an improper sequence. They are solely dependent on the relative order in which the computations occur, and can be avoided by proper scheduling of the system [Mok83].

The second kind of error is more subtle, in the sense that it is caused by violation of some specified timing constraints, such as missing deadlines. In CAPS, since a static schedule is used to execute the prototype, this problem can only happen if the MET was inaccurately specified, or if the MET was specified for running in a faster machine. What, then, is the real meaning of the MET? Is it an absolute value, or is it dependent upon the machine in which the module is running? Clearly, this is only the tip of an iceberg, and the answer is no, it cannot be absolute, since the attribute execution time is a function of the machine throughput. A module that has an MET of 150 ms for some specific machine may take longer than that to execute if running in a slower machine.

The problem is even bigger if the CAPS Software Base, which is supposed to be a collection of reusable components provided by different vendors, is taken into account. Each component should have a PSDL specification, with all the timing constraints, such as MET, MRT, MCP, etc. All of this information will be used during the execution phase of the prototype, in trying to match needs with the available components. The same problem arises regarding their timing reference, since each vendor may well have their own.

124

This discussion demonstrates the imperative need for assuming a common timing reference within CAPS. It can be anything, as long as it is consistent and used throughout the prototype. Care must be taken when choosing this reference, however, since it may lead to significant differences when dealing with reusable components from different sources.

1. Choosing a Reference

Standard measures of performance provide a basis for comparison, and *time* is the best way to measure computer performance. The computer that performs the same amount of work in the least time is the fastest. A number of popular measures have been adopted in the quest for a standard measure of computer performance, but most of them were forced into a service for which they were never intended. [HP90]

The MIPS, *million instructions per second*, is easily understood by a customer, in that faster machines means bigger MIPS. However, the MIPS measure presents the following problems:

1) MIPS is dependent on the instruction set, making it difficult to compare machines with different instruction sets

2) MIPS varies between programs on the same computer

3) MIPS can vary inversely to performance

A classic example to the third of these points is the MIPS rating of a machine with optional floating-point hardware. If it uses the hardware floating-point unit it will take less time to execute, but it will also execute fewer and more complex instructions. Software floating-point executes more but simpler instructions, resulting in a higher MIPS rating [HP90].

Another popular alternative is *million floating-point operations per second*, abbreviated as MFLOPS. However, MFLOPS is, clearly, highly dependent on the machine and on the program.

125

Other options are synthetic benchmarks, such as Whetstone and Dhrystone, but the best choice appears to be to use real programs, such as compilers, text editors, CAD tools, etc., which have inputs, outputs, and other user-defined options. [HP90]

While having a standard of performance for computers is still beyond the horizon, for prototyping purposes within CAPS, where many of the figures are still subject to change during the prototype refinement process, any of these metrics provides a good starting place. Again, for the sake of simplicity, the MIPS rating will be the reference model for performance in this work.

2. CAPS Timing Model

It will be useful to define some of the terms used in construction of the model:

CAPS Reference —Specifies the MIPS rating of a hypothetical machine, to which all of the CAPS timing information should be normalized.

HOST Reference - Specifies the MIPS rating of the host machine where CAPS is installed. This value will be automatically generated by CAPS at the start of the session, and it is the result of an Unix system call.

TARGET Reference - Specifies the MIPS rating of the target machine. In the absence of this value, it is assumed that the host machine for CAPS is identical to the target machine. This value should be provided by the user at the beginning of the design of the prototype, and will affect the retrieval of reusable components from the Software Base.

CPU Speed Ratio - Specifies the MIPS ratio between the target and the host machine. It can be changed by the user to make temporary simulations and to overcome possible timing errors. It is important to note that this value will have a very important role in debugging possible timing errors during prototype execution. Its default value is given by the formula:

$$CPUSpeedRatio = \frac{Target\ Reference}{Host\ Reference}$$

126

Table 5.1 specifies the default values which will used throughout this discussion, unless otherwise stated.

CAPS Reference	Target Reference	Host Reference	CPU Speed Ratio
10 MIPS	20 MIPS	15 MIPS	1.33

Table 5.1. Default Values for the Timing Model

a. Building the Prototype

All timing information, such as MET, PER, FW, MRT, MCP, LAT and MOP, specified by the user during the design phase of the prototype, which in most cases come from the Requirements Document, are assumed to be referenced or normalized to the Target Reference. Therefore, when, for example, defining an operator with MET = 100ms, it should be understood that 100ms would be the maximum execution time allowed for that operator if running in the target machine. It will default to the host machine if the Target Reference is not given.

Note that the MET of this operator is equivalent to 200ms with respect to the CAPS Reference; it is this value of 200ms that will be used in the query to the Software Base during the search for a matching reusable component. Observe also that this value will not affect Translation nor Scheduling, since all timing information is consistently and linearly normalized to the CAPS Reference.

b. Installing Components in the Software Base

When getting reusable components from a specific vendor or supplier, the timing reference used to classify their components should be specified along with the component. For example, when a component arrives, it should be labeled as follows: *component X has a certified MET of 100ms under a 5 MIPS machine.*

This information will allow the insertion of the component into the Software Base as a component with MET equal to 50ms, which is the correct value normalized with respect to the CAPS Reference. Note that this value will be used during its retrieval from the software base by the prototypes.

127

3. Relations between CPU Speed Ratio and Timing Errors

Assuming that all timing information from the reusable components is correct with respect to the supplier's reference, then there should be no timing errors, if the component matches the prototype specification. For example:

Suppose that a component with an MET of 120ms is needed. Then the correct query to be performed on the Software Base should be for a component with an MET of 240ms, i.e.,

$$MET_{CAPS} = MET_{TARGET} \times \frac{Target_{REF}}{CAPS_{REF}}$$

Therefore, using this component in the prototype, according to the generated static schedule, should not cause any timing errors. However, if it does cause a timing error, then it is possible to conclude that the component timing information was incorrect. To solve this problem, the following steps can be taken:

a) Increase the CPU Speed Ratio until the error disappears. This means that a reasonable MET for that component with respect to the Target reference, although it may not be the tightest one, is equal to:

$$New\ MET_{Target} = \frac{New\ CPU\ Speed\ Ratio}{Old\ CPU\ Speed\ Ratio} \times Original\ MET_{Target}$$

Note that another side effect in performing step a) is that the entire schedule is stretched, and, consequently, the slack time available for the dynamic scheduler is increased, since some of the timing critical operators don't need more time to execute.

b) Update the Software Base with the correct timing information for that component.

c) Reset the CPU Speed Ratio to its original value and take either step d), e) or f) to solve the problem.

d) If requirements permit,change the PSDL specification to allow the bigger MET found in step a). This in turn will require a whole new CAPS session, starting from a new

translation until the final compilation. Note that increasing the MET affects the load factor and may cause an unfeasible schedule.

e) Search the Software Base for another reusable component that matches the original MET. This new one may well have the correct information.

f) Create another new component or optimize the existing component. Validate its timing constraints and update the Software Base if succesfull.

g) If it is realized that a faster target processor is needed in order to cope with the requirements, then the Target Reference should be changed so that those timing errors disappear. Note that this change will only affect the CPU Speed Ratio, and as explained earlier, and will not change the schedule. Theoretically, the necessary change for the Target Reference can be derived very easily from the following formula:

New Target Reference = New CPU Speed Ratio × Original Host Reference

The other source of timing errors is found when dealing with user-created components. In other words, the component just created takes more time than that specified. For example, assume the previous situation, where a component with MET of 120ms is required. Since the host machine is slower than the target machine, the scheduling time will be linearly stretched by a factor of 1.33, that is, 1.33 × 120ms, or 159.6ms, will be allowed for the execution interval of this component. If timing errors occur, the following steps can be taken to eliminate them:

a) Increase the CPU Speed Ratio until the error disappears. This means that a reasonable MET for that component with respect to the Target reference, although it may not be the tightest one, is equal to:

$$\text{New MET}_{Target} = \frac{\text{New CPU Speed Ratio}}{\text{Old CPU Speed Ratio}} \times \text{Original MET}_{Target}$$

b) Reset the CPU Speed Ratio for its original value and take either step c) or d) to solve the problem.

c) If requirements permits, change the PSDL specification to allow the bigger MET found in step a). This in turn will require a whole new CAPS session, starting from

129

a new translation until the final compilation. Again, this change may cause an unfeasible schedule.

d) Rewrite the component trying to speed it up;

e) If it is realized that a faster target processor is needed in order to cope with the requirements, then the Target Reference should be changed so that those timing errors disappear. The required change for the Target Reference can be derived from the following formula:

New Target Reference = New CPU Speed Ratio × Original Host Reference

f) After getting rid of the timing errors, if it is decided to add the user-created component to the software base, the component should be associated with an MET_{CAPS} equal to $MET_{Target} \times \dfrac{CAPS_{REF}}{Target_{REF}}$.

4. How the CPU Speed Ratio affects Scheduling

The Static Schedule is basically a sequence of pairs of absolute values containing the start time and stop time for each instance of the time-critical operators within one harmonic block.

At the beginning, the static scheduler task calls the function TARGET_TO_HOST, which belongs to the package PSDL_TIMERS, and multiplies all those absolute time values by the CPU Speed Ratio. The net effect is that the scheduler will stretch or shrink all of the timing information related to the prototype in a linear fashion.
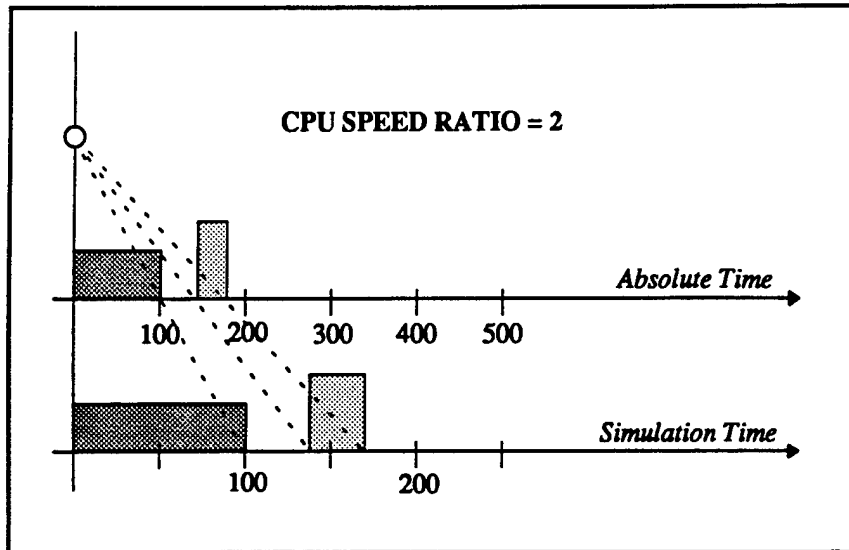
Figure 5.15. Effect of the CPU Speed Ratio on the Schedule

5. Handling Unwanted Interactions during Prototype Scheduling

A software prototyping environment needs to simulate external entities so that the entire system being prototyped can be exercised. These external entities will in most cases either generate inputs or consume outputs from the core of the system being prototyped. This requires that the timing constraints are taken into consideration during the generation of the schedule. However, it is during prototype execution that the effects are most harmful, since they will incorrectly steal CPU time from the host system. It is also unavoidable that time is spent by the host operating system to serve processes that sometimes nothing have to do with the prototype.

All these unwanted interactions can dramatically affect timing behavior and overall confidence in the prototype. The question to ask, then, is how can these timing interferences be eliminated?

To solve these problems, CAPS introduced the technique of having two different time lines. One is the absolute time line, and is driven by the real-time clock of the host machine. The second one, the simulation time, will command all the scheduling actions of the prototype.

131

What is going to happen is that whenever an external operator, or some operating system function, is being executed, the scheduling clock will be frozen, so that, for the prototype, it is as though they do not exist.

Another feature that can be explored with this technique is when an operator belonging to the prototype exceeds its scheduling interval and causes an exception. It is very likely that this will interfere with other operators, causing a chain of exceptions, when in reality, only the very first operator incurred a timing error. Because of the use of a simulated clock (the scheduling clock) it is possible to remove any excess of time from the scheduling clock, and then resume the simulation, so that no further operators will be affected.

VI. EXPERIMENTAL RESULTS

A. INTRODUCTION

Although the full implementation of the new Distributed Model is not complete, due to software limitations of the current Ada compiler technology that will be solved by the new Ada95 implementation, much can be said about expectations and also about the general scheduling capability of CAPS.

One of the biggest problems encountered during this research was the lack of an adequate set of prototypes to test the scheduler. Up to now, most of the development in CAPS has been tested with a few prototypes that may be sufficient for the development of several tools, but not for the scheduler, which requires a huge test set so that all the critical points can be exercised. This is the reason for building a PSDL random graph generator, as discussed in the next section of this chapter.

B. THE RANDOM GRAPH GENERATOR

The random graph generator has the following basic features:

1) builds PSDL prototypes with an arbitrary number of operators

2) allows the user to specify how many different prototypes are to be generated

3) provides an expert mode where the system attempts to reduce the harmonic block automatically, by changing the periods of the periodic and the transformed sporadic operators within an user-defined range

4) operates in two randomization modes: unlimited or restricted randomization

5) provides a compression capability, so that an arbitrary number of operators may be located within a bounded load factor of one. This is very useful for testing uniprocessor scheduling algorithms

6) allows the user to specify the desired percentage of timing critical operators, periodic operators, and data flow edges

7) can generate prototypes with different degrees of sparseness

8) the user can specify the maximum number of edges between two operators

9) provides a thorough scheduling information for debugging purposes

There are basically two major procedures that build the random graph. The first one is the Produce_Random_Array and the second one is the Produce_Random_Matrix. Both routines use the same data structure of the scheduler, so that the simulation is as close as possible to the real prototype.

```
type OPERATOR is
  record
    THE_OPERATOR_ID:         OPERATOR_ID := A_STRING.empty;
    THE_MET:                 MET := 0;
    THE_MRT:                 MRT := 0;
    THE_MCP:                 MCP := 0;
    THE_PERIOD:              PERIOD := 0;
    THE_WITHIN:              WITHIN := 0;
    ACTUAL_PERIOD:           PERIOD := 0;
    LOWER_PERIOD:            PERIOD := 0;
    UPPER_PERIOD:            PERIOD := PERIOD'last;
    THE_SLICES:              NODE_LIST.list := null;
    LOAD_FACT:               FLOAT := 0.0;
    FAN_IN:                  INTEGER := 0;
    FAN_OUT:                 INTEGER := 0;
  end record;

type EDGE_INFO is
  record
    ORIGIN:                  INTEGER := -1;
    DEST:                    INTEGER := -1;
    PARENT:                  INTEGER := -1;
    CHILD:                   INTEGER := -1;
    THE_LATENCY:             LATENCY := 0;
    DATAFLOW_EDGE:           BOOLEAN := false;
    OVERLAP_ABLE:            BOOLEAN := true;
    HAS_STATE_EDGE:          BOOLEAN := false;
    IMC_PER_SEC:             FLOAT := 0.0;
    IMC_INDEX:               FLOAT := 0.0;
    PR_INDEX:                FLOAT := -99.0;
    CONNECTIVITY:            INTEGER := 0;
  end record;
```

Figure 6.1. Partial View of the Data Structure Used to Build the Random Graph.

The first procedure, Produce_Random_Array, is the one that actually randomly assigns the timing constraints to the random prototype. It has two modes of operation. The first one uses a partial randomization, in the sense that only values from a pre-defined set are assigned to the timing constraints. The second mode uses a full randomization, so that any value within a finite range previously specified can be assigned.

It is in this procedure where most of the information provided by the user, such as number of prototypes to be generated, number of operators in each prototype, percentage of timing critical operators, mode of randomization, percentage of periodic operators, and compression factor are used.

In the current implementation, the restricted randomization mode generates five possible different values for MET (100, 300, 500, 700, and 1000) and four values for each of the remaining timing constraints PER, FW, MCP and MRT, which are dependent upon the previous chosen value for the MET. This was done in order to assure semantic compatibility with a valid PSDL prototype.

If one opts for unlimited randomization, then no restriction is imposed on timing constraints, rather than limiting their values within a reasonable range, which now stands between 0 and 8000 ms.

The random number generator being used has a period of approximately 2^{144}, so in order to achieve better results it is not reset after the generation of each different prototype.

The expert mode is a facility that allows the user to automatically reduce the final harmonic block length of the prototype, substantially increasing the schedulability of the prototype. For more in depth information, refer to Chapter III, Section E.

The compression factor is used so that, if the prototype happens to have a load factor bigger than one (which would mean that it couldn't run in a uniprocessor system) then the timing constraints are going to be compressed accordingly. This feature allow us to test huge prototypes for uniprocessors that otherwise, due to the random nature of the graph, would be very hard to achieve.

The second main procedure, Produce_Random_Matrix, is where artificial edges are randomly generated according to the degree of sparseness and the maximum number of edges defined by the user. It is also here where the latency for each edges is generated.

C. FIRST FINDINGS AFTER USING THE RANDOM GRAPH GENERATOR

The first finding after using the random graph generator was that the scheduling capability of the existing CAPS scheduler is very poor. It is not likely that the scheduler will find a feasible schedule for a moderate size prototype without manual adjustment of all timing constraints after a long and tedious process of trial and error. But that is not really bad because, after all, the static scheduling problem is a well known NP-Hard problem. The interesting thing, however, is that even for very small prototypes, with as few as 4 or 5 operators, and also a very limited number of edges, it still couldn't find a feasible schedule, even through the use of traditional and widely accepted algorithms, such as *earliest start time first* and *earliest deadline first,* modified for the non-preemptive case. The question to be asked is, "Why does that happen, and how can we improve it?".

After meticulous analysis of several runs, with hundreds of random prototypes, it was determined that, on average, the earliest deadline first algorithm finds a feasible schedule for prototypes with load factors less than 0.5. It was also noticeable that the schedulability of the prototype was affected somehow by the harmonic block length (HB). There were some cases where, even with load factors over 0.95, after optimizing the HB to smaller values, it was possible to find a feasible schedule, which could not be achieved with the bigger HB. The load factor definitely has a strong influence on schedulability. For the harmonic block, however, it was not thought that the influence would be so great.

There are two readily apparent explanations for the harmonic block syndrome. The first is because of the higher number of instances that can fit in a bigger HB, the probability of having two or more tasks fighting for the same time slot increases. The second explanation is partially supported by Theorem 6 in Chapter III, where it is evident that, by increasing the period of an operator, which might happen when its period is optimized, it also has an effect of increasing the probability of finding a feasible schedule.

The following problems are now apparent: First, how to decrease the load factor of our prototype, and; second, how to decrease its associated harmonic block.

The total load factor of the prototype cannot be changed much, since it comes from the user's requirements. Splitting them into multiple processors will not do much good in the current practice for non-preemptive static distributed scheduling, which requires a global schedule for the entire prototype in order to satisfy all synchronization requirements.

In order to change the harmonic block, assuming that the METs cannot be changed, it is necessary to modify the periods, but recall that they are constrained by the user's requirements. However, if we take a close look at these problems it is possible to realize that they are quite different.

Assume that the requirements allow for making little changes in the periods, which is a fair assumption, since in most of the systems it does not really matter if the period of some task is 1000 ms or 1010 ms. So the effect of such period change on the load factor

136

is clearly very small, while for the harmonic block it may represent a very big change, since it may get rid of some prime factor that was driving up the least common multiple (LCM) of the periods. Following this line of reasoning a novel technique to decrease the harmonic block was discovered, and will be described in the next section.

D.   MINIMIZING THE HARMONIC BLOCK

The need for a harmonic block comes from the fact that, unlike most of the problems in classical scheduling, this periodic task set contains an infinite number of instances. Therefore, in order to calculate a static schedule for the task set, it is necessary to find a time interval which can be repeated forever. When the completion time of the first instances are restricted to be less than or equal to the periods, it is common for the harmonic block to be the least common multiple (LCM) of the periods for such an interval. However, when those restrictions to the deadlines do not apply, it has been proven in Chapter III Section C that it is sufficient to increase the time interval to twice the LCM. At any rate, the point to be made is that in both cases the size of the LCM is critical and, for the reasons explained in the previous section, it is desirable to make it as small as possible.

Formally, the least common multiple of two natural numbers i and j is the smallest natural number that is divisible by both i and j. It is also known from Number Theory that every positive integer can be written uniquely as the product of primes, where the prime factors are powers of some positive integer.

From the above definitions, it can clearly be seen that the LCM of two natural numbers i and j will have in its prime factorization all of the prime factors of the original numbers raised to the maximum exponent, as shown in the following example.

Example:

$$i = 120 \qquad = 2^3 \times 3 \times 5$$
$$j = 100 \qquad = 2^2 \times 5^2$$
$$\text{LCM } (i,j) \qquad = 2^3 \times 3 \times 5^2 = 600$$

This same approach can be extrapolated to a case where several numbers are present, instead of only two. So now the problem is decreasing the LCM of a set of periods.

There are two basic approaches. The first one is trying to decrease the factor with the biggest prime, and the other is decreasing the biggest prime factor. Clearly, the second approach is more expedient, but still leaves the following problem. Suppose all of the periods which are contributing for the factors in the LCM are identified, and have been placed into a critical list, with some kind of mapping to the factors they are affecting. Now, assume that the period which is contributing for the biggest factor is changed. With luck, that biggest factor may be eliminated. However, the exponent of some other prime factor from that same period may be increased, now becoming the critical one for the LCM. In other words, it is necessary to re-evaluate the critical list and the corresponding mapping after each iteration of the optimization process, or one may end up with a non-optimal solution.

After this brief description of the problem statement, it is possible to introduce the algorithm for optimizing the LCM, which is presented in Figure 6.2.

```
Algorithm Optimize_LCM
        For every period calculate its prime factors;
        Calculate the initial LCM for the periodic task set and its prime factors;
        Set the flag LCM is decreasing to false;
        While there exists a prime factor of the LCM not yet optimized loop
                Insert those tasks whose periods are contributing for the LCM factors into the
                Critical List in decreasing order of their contribution. In other words, the head of
                the Critical List will be the task with the biggest contribution to the LCM;
                While the Critical List is non-empty loop
                        Pick the task which is in the head of the Critical List;
                        Remove its contribution from the LCM;
                        For each period within its allowable range loop
                                Calculate the new LCM;
                                If LCM is decreasing then record this period as the best one so far;
                        end loop;
                        If LCM is decreasing then
                                Update the new LCM and the task prime factors
                        end if;
                        Remove this task from the Critical List;
                end loop;
                if LCM is decreasing then
                        It means that come critical task in the Critical List had its period changed
                        and consequently reduced the LCM. Now is the subtle part, even if we had
                        some period in the Critical List that couldn't have its biggest factor
                        changed, so that the LCM could decrease, it needs to be reconsidered, since
                        the order in which the Critical List was scanned matters!! In other words,
                        after all the others in the Critical List have been processed, it may well now
                        be possible to change that same task so that the LCM will be decreased. So,
                        we need to calculate the new LCM and start all over again.
                else if LCM is not decreasing
                        Means that none of the critical tasks in the Critical List were able to get rid
                        of their biggest factor, and so there is nothing else to do other than skip to
                        the second biggest factor, and so forth.
                end if;
                Set LCM decreasing flag back to false;
        end loop;
end Algorithm Optimize_LCM;
```

Figure 6.2. Algorithm for Optimizing the LCM

Although its optimality has not been formally proven, it is believed that this algorithm will always lead to near-optimal results. By applying this algorithm to some random task sets it was possible to tremendously reduce the harmonic block, with some positive effects in schedulability. It should also be noted, by the examples shown below,

that the periods are of critical importance. With very few changes in the periods an enormous decrease in the LCM can be achieved, with consequently few effects on the load factor.

Example	INITIAL PERIOD	NEW PERIOD
	100	100
	500	500
1	600	600
	800	750
	1033	1000
LCM	12,396,000	3000

4132 TIMES SMALLER

Example	INITIAL PERIOD	NEW PERIOD
	1500	1500
	1320	1400
	1677	1820
	500	500
2	700	700
	800	875
	1000	1000
	999	1092
	2987	3250
LCM	5.13763486E+14	273,000

1,881,917,532 TIMES SMALLER

Figure 6.3. Optimization Results

E. THE NEW DISTRIBUTED SCHEDULING ALGORITHM - SOME RESULTS

After running several hundreds prototypes with typical values for the timing constraints (such as MET, MRT, MCP and PER) it was possible to make several conclusions in addition to those already cited in the previous sections. One of them, and actually the main driving force for directing us to distributed scheduling, was the palpable necessity for prototypes with load factors bigger than 1.0, specifically in our applications domain.

Another major point discovered after this research is the real need for supporting and advising the real-time system designer, mainly with respect to the values for the timing constraints. Remember that non-preemptive static scheduling is a well known NP-Hard

140

problem, so that unless P=NP, there is not much hope of finding better ways to solve this problem. That is why, sometimes, in prototypes with only two nodes, it was impossible to find a feasible schedule.

So, what is really needed is to find better ways to live with this problem. One of the ways to accomplish this is by providing better support in the area of schedulability tests, which is also a known NP-Hard problem. That is why several theorems were presented in Chapter III, which, it is hoped, will help in finding and pin-pointing some of the problems in the user's design.

It is possible by making use of those theorems to suggest changes in the timing constraints of a set of tasks, or even in a specific task, to suggest different partitions so that some tasks are kept together due to the similarities of their timing constraints, etc.

Now the scheduler can handle prototypes with load factors bigger than one, by applying the allocation algorithm described in Chapter V. The user can either specify the maximum load factor allowed per processor, or the number of processors. It is also capable of generating a schedule, if one can be found, by using a distributed version of the Earliest Deadline First algorithm. By making use of the Fundamental Synchronization Theorem it is now possible to divide the schedule into several smaller schedules, so that its complexity is tremendously decreased.

The robustness of the new scheduler is enhanced due to the large testing that was made possible by the random graph generator. Several important bugs were found during these experiments. It was possible to analyze and compare the performance of the different uniprocessor scheduling algorithms currently implemented in CAPS. The output generated by the scheduler is now more comprehensive, improving the debugging capability.

An expert mode is provided to the designer, so that the harmonic block will be decreased with some effects on the load factor. A possible enhancement for the expert mode is to combine it with the actual scheduling. In other words, instead of applying the

optimization algorithm to the entire task set in only one step, prior to the scheduling, an attempt should be made to schedule the task set after each optimization iteration.

As can be seen, quite a lot has been accomplished towards a more dependable and reliable scheduler, but much more needs to be done so that CAPS can become a true design aid to real-time system designers.

VII. CONCLUSIONS AND RECOMMENDATIONS

A. SUMMARY OF THE DISSERTATION

This dissertation can be roughly divided into three parts. The first part (Chapters I through III) presents a review of the most important results in the area of hard real-time scheduling and introduces several theorems to improve the schedulability analysis of task sets containing both periodic and sporadic tasks. The effects of precedence relationships among the tasks on these theorems is also analyzed. Although most of the work was done for the non-preemptive model, several results are also applicable to the preemptive case, as highlighted throughout the dissertation. The second part of the dissertation (Chapter IV) introduces the novel method of hard real-time distributed scheduling without explicit synchronization. The motivation for this new approach is the complexity of the hard real-time scheduling problem, where for even small size systems running in a uni-processor environment, it is extremely hard to find a feasible schedule. With the addition of one more variable, such as distributed processing, the general scheduling problem becomes intractable, and unless P=NP, there is no reason to foresee any solution to this problem. It was therefore decided to sacrifice timing constraints in order to decrease the complexity of the scheduling problem. Depending on the application, this approach may not be applicable. However, this approach should work in most cases, especially in prototyping, which is usually in the early stages of the life cycle of the system, allowing for the fine tuning of timing requirements. The third part of the dissertation deals with the architectural aspects of implementation of a distributed real-time scheduler without making use of any explicit synchronization. The following paragraphs present a summary of the salient results found in each chapter.

Chapter I highlights the increasing demand for real-time systems in life-critical areas that were heretofore unexplored. Some basic definitions for hard real-time systems are also introduced, and a taxonomy for scheduling is proposed. Past research in real-time scheduling is reviewed and the major results are listed in tabular form. A brief note shows

that the complexity of scheduling algorithms for a non-periodic task set, which are solved in polynomial time, become exponential when dealing with periodic task sets. Some complexity results for message routing in hard real-time distributed systems are also presented.

Chapter II presents a brief discussion of the Computer Aided Prototyping System (CAPS) which is a software engineering tool for developing prototypes of real-time systems. The Prototyping System Description Language (PSDL) and its facilities for modeling real-time systems are also described in this chapter.

Chapter III formalizes the real-time scheduling problem for periodic and sporadic task sets. It starts by introducing the scheduling model that will be used throughout the dissertation, and proceeds with the presentation of several theorems for improving the schedulability analysis of tasks with hard deadlines. The three most important results in this chapter are established by Theorems 6, 7, and 8. The Task Demand Theorem (Theorem 6), specifies necessary conditions for task sets with arbitrary deadline and release times to be schedulable. It is also shown that if release times are taken into consideration, due to precedence relations, for example, the conditions are no longer necessary, but only sufficient. Theorem 7 extends this result, and proves that any periodic or sporadic task set satisfying the conditions of Theorem 6 can be scheduled with the Earliest Deadline First (EDF) algorithm, thus making the conditions specified in Theorem 6 necessary and sufficient. The Harmonic Block Theorem (Theorem 8) introduces the novel concept of transient and cyclic schedules, which is an enhancement of the traditional method for calculating a cyclic schedule, if one exists. It is shown by example that this latter method improves the schedulability of task sets which were found to have no feasible schedule by the traditional method. Later in the chapter all previous results are re-analyzed for the case where precedence relationships exist among the tasks. Theorem 8 is also extended to handle the situation where latencies are involved in the scheduling. Note that the net effect of introducing latencies in the problem is that the schedule can no longer be assumed to have no inserted idling time in the interval [0,LCM]. Finally, a

methodology to convert sporadic operators into equivalent periodic ones is presented, along with some important considerations about this conversion.

Chapter IV presents an in-depth discussion covering all possible aspects of the communication involving two PSDL operators connected by some kind of data stream. The synchronization problem between producers and consumers is carefully analyzed, as is the underlying meaning of missing a deadline within the context of a real-time system. The conclusion reached is that missing deadlines are always attached to data that is not generated or consumed in the proper timing. This *data approach* for the synchronization problem will lead to the new distributed scheduling model with no explicit synchronization, which is formalized by the Fundamental Synchronization Theorem (Theorem 9). The application of this theorem allows each set of tasks allocated to a particular processor to be treated as a totally independent set, provided that some more stringent timing constraints are satisfied. This approach will greatly decrease the scheduling complexity of large distributed real-time systems, although it may be applicable as well to cases involving uni-processors or shared memory multiprocessors. At the end of this chapter are some considerations about the allocation model implemented for the distributed scheduler in CAPS.

Chapter V presents the current implementation of the CAPS uni-processor scheduler and it also proposes an architecture for implementing the full version of the distributed scheduler. It describes two options for implementing the distributed version. The first is to use the currently available C libraries for implementing the communication sub-system. Several problems with this approach are also addressed. The second option relies on the availability of a full Ada95 compiler, which, according to the Ada95 Reference Manual's Annex E, will support communications between tasks running in different processors. In the last section of this chapter several interesting considerations are presented regarding the timing problems involved in a typical software prototyping environment. Topics such as simulated time, normalized reference for time information, timing errors, and why they happen are covered in this section.

145

Chapter VI presents experimental results of the partially implemented distributed scheduler in CAPS. The random PSDL graph generator, which was one of the important factors for a better understanding of the scheduling problems in CAPS, is described. Finally, an important issue is discussed which is not given enough attention by most of researchers, namely, the least common multiple (LCM) of the periods of a periodic task set, which ultimately will determine the size of the cyclic schedule for the task set. It is demonstrated that, by making minor changes in the original periods, the final LCM and, consequently, the solution space of the corresponding scheduling problem can be drastically reduced.

Chapter VII is the conclusion, but it also proposes some modifications for CAPS, so that it can become a more dependable and reliable design tool for building real-time systems.

B.   POSSIBLE CAPS MODIFICATIONS

As a result of this dissertation, several weaknesses and areas requiring improvement within the entire CAPS and PSDL were identified. Many errors in the static scheduler were corrected, but others require further effort.

1.   Enhancing the CAPS Syntax Directed Editor (SDE)

As discussed in Chapter IV, several semantic checks for the input PSDL program are currently enforced by the scheduler. It seems reasonable, however, to allow most of these checks to be enforced by the SDE. This approach would allow the user to detect and receive warnings about the design in the early stages of prototyping. In doing so, the designer would not have to go all way back to the SDE when a semantic error was found by the scheduler.

2.   Tasks with Soft Deadlines

In CAPS there are only tasks with hard deadlines (TC), or tasks with no deadlines at all (NTC). In real-time systems however, there are often a third kind of deadline, but if it is missed for some reason it does not cause any harm to the system. This is known as a

146

"soft deadline". Right now for example, an NTC operator can starve for a long time before its execution. This was certainly not the intention of the designer when the operator was placed in the prototype. This anomaly happens because the Non-Time Critical operator (NTC) depends on the time left by the static scheduler, which can be none if the load factor is 1.0, and all the TC operators use their entire MET.

The implementation of tasks with soft deadlines or some other approach, like the time-value functions presented in Chapter I, would greatly improve the scheduling capability of prototypes in CAPS.

3. Preemptive Static Scheduling

So far this option has not been used in CAPS because of the ADA83 tasking model, which prevents tasks with higher priority to change their relative position in the FIFO queue of a rendezvous. ADA95 however, allows dynamic changes in the queue according to their priority and, therefore, the preemptive model again becomes a valid and reasonable option for the CAPS scheduler. Note that, in general, the preemptive scheduling problem is easier to deal with than the non-preemptive one, allowing much better scheduling results. Further research is needed, but it appears that allowing a mixture of preemptive and non-preemptive tasks is the best approach available.

4. Triggering Conditions versus Stream Types

Currently, in the PSDL model a sampled stream does not guarantee that the data is not lost or replicated. In the same model, however, the stream type is determined from the triggering condition of the consumer operator, e.g., an operator with a TRIGGERED BY SOME condition is supposed to guarantee that its output is based on the most recent value of the input sampled stream, which is to some extent a contradiction. Our suggestion is to separate triggering conditions from the type of the streams, so that there can be a more orthogonal grammar for PSDL. A sampled stream should be defined as the stream where the data can be read zero or more times, whether in a data flow stream it can be read once and only once. It is understood that this definition better conveys the real

147

meaning of a stream, since a stream by itself should not guarantee whether or not the data is lost; the stream is simply a mechanism to transfer data.

Once the idea of separating triggering conditions from stream types is accepted, it is necessary to check which are the valid combinations. These combinations are presented in Table 7.1, and should be considered valid unless an exception is noted.

	TRIGGERED BY ALL	TRIGGERED BY SOME	NO TRIGGER
DATA FLOW STREAM	OK	NOK (2)	NOK (3)
SAMPLED STREAM	NOK (1)	OK	OK

Table 7.1. Triggering Condition and Stream Type Combinations

(1) Assume an operator A TRIGGERED BY ALL X,Y, where X and Y are sampled streams. Suppose data arrived only in X. It is necessary to wait for new data in Y, but after A is fired, both pieces of data are consumed, and the old data cannot be used again, otherwise it is impossible to know which data is new or old, and therefore the existence of this case does not make sense. The only situation where this combination would be needed is if combinations of TRIGGERED BY SOME and TRIGGERED BY ALL are allowed to exist for the same operator. Note, however, that this combination can always be implemented in two steps and with one additional operator.

(2) Assume an operator A TRIGGERED BY SOME X,Y, where X and Y are data flow streams. Suppose only X gets new data. Operator A will fire and consume the data in X, leaving nothing behind because it is data flow. When new data comes in Y, there is nothing in X, and an underflow will occur.

(3) It does not make sense, because if there is no trigger, how can the consumer be guaranteed to always catch new data that comes into the data flow?

5. Estimating the Execution Time

As explained earlier, the MET is an upper-bound on the execution time of an operator, and it is this value which is used by the scheduler to generate the static schedule. Therefore, everything that can be done to decrease the MET is going to have a direct effect on the schedulability of the prototype. It would be nice if it were possible to, at run-

148

time, keep track of the real amount of time needed by each operator, so that feedback could be given to the user about its real MET for further update of the Software Base.

6. The Uninitialized Sampled Stream Problem

Suppose there is a non-time critical operator (NTC) connected to a time critical operator (TC) by a sampled stream. Clearly, the TC operator may be fired at least once before the NTC operator, and therefore it will read garbage from the sampled stream.

This problem is aggravated in distributed scheduling, as shown by the example in Figure 7.1.
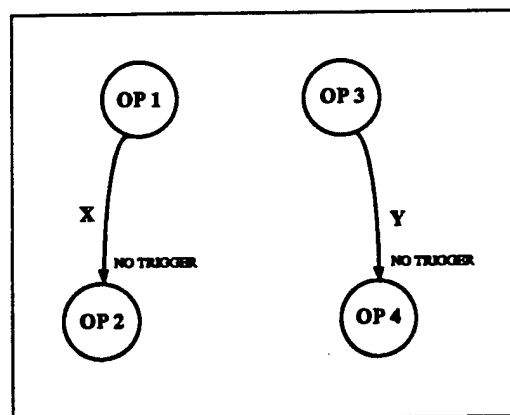
Figure 7.1. The Uninitialized Sampled Stream Problem

Note that this example does not cause any problem in the uni-processor case, but in distributed scheduling, if OP_1 and OP_2 are assigned to different processors, OP_2 may fire before OP_1, and an uninitialized sampled stream will be read. A proposed solution would be to force the sampled stream to be declared as a state stream whenever an initial value is needed.

7. State Stream versus Data Flow

It does not make sense to have an operator TRIGGERED BY ALL X, if X is, for example, a state stream. The reason for this is that values carried by state streams should always be available, and in a data flow stream the value is consumed after it is read, and no longer available. A warning should therefore be given if this happens in a PSDL program.

149

C. CONCLUSIONS

This dissertation shows that hard real-time systems and, more specifically, hard real-time scheduling, are areas which are far from being totally explored. The next generation of hard real-time systems will be extremely large, complex, and most certainly distributed. They will be *truly* distributed, without any need for synchronization among processors.

Most of the work so far in this area has been concentrated on finding better scheduling algorithms, without concentrating on the real need for synchronization. Deadlines are always attached to data not being generated or consumed in a timely fashion. This dissertation is the first work ever done in the area of distributed scheduling without any explicit synchronization, and it is hoped that it will mark a turning point in the distributed scheduling field. It is far from being complete, but it does provide a totally different perspective on the distributed scheduling problem.

Finally, this dissertation offers the following scientific contributions:

1) A new model for distributed scheduling without synchronization;

2) Several theorems on the schedulability of periodic and sporadic task sets, improving the state of the art in the scheduling field;

3) A general Timing Model for Pototyping Systems, which will enable interaction with different time references, keeping total consistency throughout the design;

4) A method for optimizing the schedule length of periodic task sets. This approach will decrease the time spent in scheduling and improve the chances of finding a feasible schedule;

5) Making use of recent theoretical results in scheduling, they have been adapted to the model in this work in order to support a systematic and formal method for the design, synthesis, and validation of timing constraints in hard real-time systems.

More specifically related to CAPS, the following contributions can be listed as additional results of this dissertation:

1) Enhancement of the existing CAPS Prototyping System with a new Distributed Scheduler with:

- allocation capability
- increased reliability
- better schedulability
- and an expert mode

2) A Random PSDL Graph Generator.

LIST OF REFERENCES

[AB93]   N. Audsley and A. Burns, *Real-Time System Scheduling*, Technical Paper University of York, UK, 1993.

[Ada95]   *Ada 95 Reference Manual*, Intermetrics, Inc., January 1995.

[Bad93]   S. Badr, *A Model and Algorithms for a Software Evolution Control System*, Ph.D. Dissertation, Naval Postgraduate School, December 1993.

[Bak74]   K. Baker, *Introduction to Sequencing and Scheduling*, John Wiley & Sons, Inc., 1974.

[BFR71]   P. Bratley, M. Florian and P Robillard, *Scheduling with Earliest Start and Due Date Constraints*. Naval Research Logistics Quarterly, 18(4), December 1971.

[BL91]   V. Berzins and Luqi, *Software Engineering with Abstractions*, Addison-Wesley, Reading, MA, 1991.

[Bla76]   J. Blazewicz, *Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines*, Proceedings of the International Workshop on Modelling and Performance Evaluation of Computer Systems, Amsterdam, North-Holland, pp.57-65, 1976.

[Boa84]   B.H. Boar, *Application Prototyping: A Requirements Definition Strategy for the 80's*, John Wiley and Sons, Inc., New York, 1984.

[Boe86]   B.W. Boehm, *A Spiral Model of Software Development and Enhancement*, ACM SIGSOFT Software Engineering Notes, vol. 11, no. 4, pp. 14-26, August 1986.

[Boo87]   G. Booch, *Software Engineering with Ada*, 2nd ed., Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1987.

[Bro94]   J. Brockett, *The Computer-Aided Prototyping System (CAPS) Tutorial*, Naval Postgraduate School, November 1994.

[BS74]   K.R. Baker and Z.S. Su, *Sequencing With Due-dates And Early Start Times To Minimize Maximum Tardiness*, Naval Research Logistics Quarterly, vol. 21, pp. 171-176, 1974.

[BT83]   J.A. Bannister and K.S. Trivedi, *Task Allocation in Fault-Tolerant Distributed Systems*, Acta Informatica, Springer-Verlag, 1983.

153

[CG72]   E.G. Coffman and R. Graham, *Optimal Scheduling for Two-Processor Systems*, Acta Informatica, 1, 1972.

[CHL80]   W.W. Chu, L.Y. Holloway, M.T. Lan and K. Efe, *Task Allocation in Distributed Data Processing*, Computer, vol. 13, no. 11, pp. 57-69, November 1980.

[CSR87]   S.C. Cheng,, J.A. Stankovic and K. Ramamritham, *Scheduling Algorithms for Hard Real-time Systems - A Brief Survey*, COINS Technical Report 87-55, June 10, 1987.

[Dam94]   D. Dampier, *A Formal Method for Semantics-Based Change-Merging of Software Prototypes*, Ph.D. Dissertation, Naval Postgraduate School, June 1994.

[DD86]   S. Davari and S.K. Dhall, *An on Line Algorithm for Real-Time Tasks Allocation*, IEEE Real-Time Systems Symposium, December 1986.

[DL78]   S.K. Dhall and C.L. Liu, *On a Real-Time Scheduling Problem*, Operations Research, vol. 26, no. 1, pp. 127-140, February 1978.

[Dol93]   S. Dolgoff, *Automated Interface for Retrieving Reusable Software Components*, Master Thesis, Naval Postgraduate School, September 1993.

[EFM83]   J. Erschler, G. Fontan, C. Merce and F. Roubellat, *A New Dominance Concept in Scheduling N Jobs on a Single Machine with Ready Times and Due Dates*, Operations Research, 31(1), 1983.

[GJ75]   M.R. Garey and D.S. Johnson, *Complexity Results for Multiprocessor Scheduling Under Resource Constraints*, SIAM Journal of Computing, 1975.

[GJ77a]   M.R. Garey and D.S. Johnson, *Two-processors Scheduling with Start-times and Deadlines*, SIAM Journal on Computing, vol. 6, pp. 416-426, 1977.

[GJ77b]   T. Gonzalez and D.B. Johnson, *A New Algorithm for Preemptive Scheduling of Trees*, Technical Report 222, Computer Science Department, Pennsylvania State University, 1977.

[GJS81]   M.R. Garey, D.S. Johnson, B.B. Simons and R.E. Tarjan, *Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadlines*, SIAM Journal Comput., 10(2), pp. 256-269, May 1981.

[Hor74]   W.A. Horn, *Some Simple Scheduling Algorithms*, Naval Research Logistics Quarterly, 21, pp. 177-185, 1974.

[HP90]   J. Hennessy and D. Patterson, *Computer Architecture a Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

[Hu61]   T.C. Hu, *Parallel Scheduling and Assembly Line Problems*, Operations Research, 9, pp. 841-848, 1961.

[Jac55]   J.R. Jackson, *Scheduling a Production Line to Minimize Maximum Tardiness*, Research Report 43, Management Science Research Project, University of California, Los Angeles, 1955.

[Jef89]   K. Jeffay, *The Real-Time Producer/Consumer Paradigm: Towards Verifiable Real-Time Computations*, Ph.D. Thesis, University of Washington, Department of Computer Science, Technical Report #89-09-15, September 1989.

[Jen77]   C.J. Jenny, *Process Partitioning in Distributed Systems*, Digest of Papers National Telecommunications Conf., 1977.

[JSM91]   K. Jeffay, D. Stanat and C. Martel, *On Non-Preemptive Scheduling of Periodic and Sporadic Tasks*, Proceedings of Real-Time Systems Symposium, December 1991.

[Lam88]   D.A. Lamb, *Software Engineering Planning for Change*, Prentice Hall, Englewood Cliffs, NJ, 1988.

[Law73]   E.L. Lawler, *Optimal Sequencing of a Single Machine Subject to Precedence Constraints*, Management Science, 19, pp. 544-546, 1973.

[LB88]   Luqi and V. Berzins, *Rapidly Prototyping Real-Time Systems*, IEEE Software, vol. 5, pp. 25-36, 1988 and Technical Report NPS52-87-005, Naval Postgraduate School, Monterey, CA, 1987.

[LBY88]   Luqi, V. Berzins and R.T. Yeh, *A Prototyping Language for Real-time Software*, IEEE Transactions on Software Engineering, vol. 14, no. 10, pp. 1409-1423, October 1988.

[Lei80]   D. Leinbaugh, *Guaranteed Response Times in a Hard Real-Time Environment*, IEEE Trans. on Software Engineering, vol. SE-6, pp. 85-91, 1980.

[LK88]   Luqi and M. Ketabchi, *A Computer Aided Prototyping System*, IEEE Transactions on Software Engineering, March 1988 and IEEE Software, vol. 5, pp. 66-72, March 1988.

[LL73]   C.L. Liu and J.W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment*, Journal of the ACM, vol. 20, no. 1, pp. 46-61, January 1973.

[LLK76]   B.J. Lageweg, J.K. Lenstra and A.H.G. Rinnooy Kan, *Minimizing Maximum Lateness on One Machine: Computational Experience and Some Applications*, Statistica Neerlandica 30, pp. 25-41, 1976.

[LLS91]   J.W.S. Liu, K.J. Lin, W.K. Shih, A.C. Yu, J.Y. Chung and W. Zhao, *Algorithms for Scheduling Imprecise Computations*, IEEE Computer, pp. 58-68, May 1991.

[LS86]   J.P. Lehoczky and L. Sha, *Performance of Real-Time Bus Scheduling Algorithms*, ACM Performance Evaluation Review, Special Issue, vol. 14, no. 1, May 1986.

[LSB93]   Luqi, M. Shing and J. Brockett, *Real-Time Scheduling in System Prototyping*, Proc. Fourth International Workshop on Rapid System Prototyping, Research Triangle Park, NC, pp. 28-30, June 1993.

[LSD89]   J.P. Lehoczky, L. Sha and Y. Ding, *The Rate Monotone Scheduling Algorithm: Exact charactherization and average case behavior*, Proceedings of IEEE 10th Real-Time Systems Symposium, pp. 166-171, December 1989.

[LTW89]   J.Y. Leung, T.W. Tam, C.S. Wong and G.H. Young, *Routing Messages with Release Time and Deadline Constraints*, Proc. of Euromicro Workshop on Real Time, Como, Italy, pp. 168-177, 1989.

[Luq89]   Luqi, *Software Evolution Through Rapid Prototyping*, IEEE Computer, pp. 13-25, May 1989.

[Luq93]   Luqi, *Real-Time Constraints in a Rapid Prototyping Language*, Computer Language, vol. 18, no. 2, pp. 77-103, 1993.

[LW90]   J.Y. Leung and C.S. Wong, *Minimizing the Number of Late Tasks with Error Constraint*, Proc. of the 11th IEEE Real-Time Systems Symposium, pp. 32-40, 1990.

[LY82]   D.W. Leinbaugh and M.R. Yamini, *Guaranteed Response Times in a Distributed Hard Real-Time Environment*. Proc. IEEE Real-Time Systems Symp., December 1982.

[Mar82]   C. Martel, *Preemptive Scheduling with Release Times, Deadlines, and Due Times*, J. ACM, 29(3), 1982.

[MC70]   R.R. Muntz and E.G. Coffman, *Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems*, J. ACM, 17(2), pp. 324-338, April 1970.

[Mok76]   A.K. Mok, *Task Scheduling in the Control Robotics Environment*,TM-77, Laboratory for Computer Science, MIT, September 1976.

[Mok83]   A.K. Mok, *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1983.

[Moo68]   J. Moore, *An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs*, Management Science, vol. 15, no. 1, pp. 102-109, September 1968.

[Pre87]   R.S. Pressman, *Software Engineering: A Practitioners Approach*, 2nd. ed., McGraw-Hill, Inc., New York, NY, 1987.

[Sch90]   S.R. Schach, *Software Engineering*, Aksen Associates, 1990.

[Sim83]   B. Simons, *Multiprocessor Scheduling of Unit-Time Jobs with Arbitrary Release Times and Deadlines*, SIAM Journal for Computing, 12(2), pp. 294-299, May 1983.

[SR88]   J.A. Stankovic and K. Ramamritham. *Tutorial on Hard Real-Time Systems*, IEEE Computer Society Press, Washington, DC, 1988.

[SSN93]   J.A. Stankovic, M. Spuri, M. Di Natale and G. Buttazzo, *Implications of Classical Scheduling Results for Real-Time Systems*, CMPSCI Technical Report 93-23, March 1993.

[Sun90]   *Network Programming Guide*, Sun Microsystems, Inc., 1990.

[SW89]   S.M. Shatz and J. Wang, *Tutorial: Distributed Software Engineering*, IEEE Computer Society Press, 1989.

[Tae93]   *TAE+ Reference Manual*, Century Computing, Inc., September 1993.

[Ull75]   J.D. Ullman, *NP-Complete Scheduling Problem*, Journal of Computer and System Sciences, vol. 10, pp. 384-393, 1975.

[Ull76]   J.D. Ullman, *Complexity of Sequence Problem*, in E.G. Coffman, Computer and Job-Shop Scheduling Theory, John Wiley & Sons, NY, 1976.

[XP90]   J. Xu and D. Parnas, *Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations*, IEEE Transactions on Software Engineering, vol. 16, no. 3, pp. 360-369, March 1990.

[You89]   E. Yourdon, *Modern Structured Analysis*, Yourdon Press, Englewood Cliffs, NJ, pp. 80-95, 1989.

[ZLC94]   J. Zhu, T.G. Lewis and J. Colin, *Scheduling Hard Real-Time Constrained Tasks on One Processor*, To be published.

BIBLIOGRAPHY

[BDW86] J. Blazewicz, M. Drabowski, and J. Weglarz, *Scheduling Multiprocessor Tasks to Minimize Schedule Length*, IEEE Transactions on Computer, C-35(5), 1986.

[BS93] G. Buttazzo and J.A. Stankovic, *RED: A Robust Earliest Deadline Scheduling Algorithm*, submitted to IEEE Transactions on Computers, March 1993.

[BSR88] S. Biyabani, J.A. Stankovic, and K. Ramamritham, *The Integration of Deadline and Criticalness in Hard Real-Time Scheduling*, Proceedings of the Real-Time Systems Symposium, December 1988 and IEEE Transactions on Software Engineering, 1988.

[CC89] H. Chetto and M. Chetto, *Scheduling Periodic and Sporadic Tasks in a Real-Time System*, Information Processing Letters vol. 30, no. 4, pp. 177-184, February 1989.

[Cer89] J.J. Cervantes, *An Optimal Static Scheduling Algorithm for Hard Real-Time Systems Specified in a Prototyping Language*, Master's Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, December 1989.

[Cha92] T.C. Chang, *Static Scheduler for Hard Real-Time Tasks on Multiprocessor Systems*, Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, September 1992.

[CSR86] S. Cheng, J. Stankovic, and K. Ramamritham, *Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real-Time Systems*, IEEE Real-Time Systems, Symposium, December 1986.

[Efe82] K. Efe, *Heuristic Models of Task Assignment Scheduling in Distributed Systems*, IEEE Computer, June 1982.

[Fan90] B. Fan, *Evaluations of Some Scheduling Algorithms for Hard Real-Time Systems*, Master's Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, June 1990.

[GJ79] M.R. Garey and D.S. Johnson, *Computers and Intractability; A guide to the Theory of NP-Completeness*, Freeman; San Francisco, 1979.

[Gra76] R. Graham, *Bounds on the Performance of Scheduling Algorithms, chapter in Computer and Job Shop Scheduling Theory*, John Wiley and Sons, pp. 165-227, 1976.

[HL88]   K. Hong and J. Y-T Leung, *On-line Scheduling of Real-Time Tasks*, IEEE 1988.

[HS91]   W. Halang and A. Stoyenko, *Constructing Predictable Real-Time Systems*, Kluwer Academic Publishers, 1991.

[Hsu90]   L. Hsu, *Multiprocessor Scheduling for Hard Real-Time Software*, Master's Thesis, Computer Science, Naval Postgraduate School, Monterey, CA, June 1990.

[JL88]   Janson, D.M. and Luqi, *A Static Scheduler for the Computer Aided Prototyping System*, Proceedings of the 3rd. Annual COMPASS Conference, Gaithersburg, MD, pp.92-97, July 1988.

[Lev91]   J. Levine, *Efficient Static Schedulers for the CAPS Systems*, Master's Thesis, Computer Science Department, Naval Postgraduate School, Monterey, CA, June 1991.

[LRD93]   J. W.S. Liu., J.L. Redondo, Z. Deng, T-S. Tia, R. Bettati, A. Silberman, M. Storch, R. Ha and W-K. Shih, *PERTS: A Prototyping Environment for Real-Time Systems*, University of Illinois at Urbana, Technical Report UIUCDCS R-93-1802, May 1993.

[Luq89]   Luqi, *Handling Timing Constraints in Rapid Prototyping*, IEEE Transactions on Software Engineering, 1989 and in proceedings of the 22nd Annual Hawaii International Conference on System Science, Kailua-Kona, HI, January 1989.

[Red93]   J.L. Redondo, *Schedulability Analyser Tool*, University of Illinois at Urbana, Technical Report UIUCDCS R-93-1791, February 1993.

[SHH91]   A. Stoyenko, V. Hamacher and R. Holt, *Analyzing Hard Real-Time Programs for Guaranteed Schedulability*, IEEE Trans. on Software Engineering, vol. SE-17, pp. 737-750, 1991.

[Shi91]   Man-Tak Shing, *Efficient Scheduling Algorithms for Rapid Prototyping of Hard Real-Time Systems*, paper, Naval Postgraduate School, Monterey, CA, May 1991.

[Sil92]   A. Silberman, *Task Graph Model*, University of Illinois at Urbana, September 1992.

[SRC85]   J.A. Stankovic, K. Ramamritham, and S. Cheng, *Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems*, IEEE Transactions on Computers, vol. C-34, no. 12, pp. 1130-1143, December 1985.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center...2
    Cameron Station
    Alexandria, VA 22304-6145

2. Dudley Knox Library 2
    Code 52
    Naval Postgraduate School
    Monterey, CA 93943-5101

3. Chairman, Department of Computer Science...1
    Code CS
    Naval Postgraduate School
    Monterey, CA 93943-5100

4. Computer Technology Programs..1
    Code 37
    Naval Postgraduate School
    Monterey, CA 93943

5. Center for Naval Analysis...1
    4401 Ford Avenue
    Alexandria, VA 22302-0268

6. Prof. Man-Tak Shing 10
    Code CS/Sh
    Naval Postgraduate School
    Monterey, CA 93943

7. Prof. Luqi 10
    Code CS/Lq
    Naval Postgraduate School
    Monterey, CA 93943

8. Prof. Amr Zaky...1
    Code CS/Za
    Naval Postgraduate School
    Monterey, CA 93943

16. Captain David A. Dampier. 1
     Software Technology Branch
     Army Research Laboratory
     115 O'Keefe Building
     Georgia Institute of Technology
     Atlanta, GA 30332-0800

17. Gabinete do Ministro da Marinha 1
     A/C Brazilian Naval Commission
     4706 Wisconsin Ave., N.W.
     Washington, DC 20016

18. Estado Maior da Armada 1
     A/C Brazilian Naval Commission
     4706 Wisconsin Ave., N.W.
     Washington, DC 20016

19. Instituto de Pesquisas da Marinha 3
     Diretor
     A/C Brazilian Naval Commission
     4706 Wisconsin Ave., N.W.
     Washington, DC 20016

20. Instituto de Pesquisas da Marinha 1
     Grupo de Sistemas Digitais
     Rua Ipiru 2, Ilha do Governador,
     Rio de Janeiro, BRAZIL   21931

21. Diretoria de Armamento e Comunicacoes da Marinha 1
     A/C Brazilian Naval Commission
     4706 Wisconsin Ave., N.W.
     Washington, DC 20016

22. Diretoria de Ensino da Marinha... 1
     A/C Brazilian Naval Commission
     4706 Wisconsin Ave., N.W.
     Washington, DC 20016

23. Pontifícia Universidade Católica... 1
     Depto. de Informática
     R. Marquês de São Vicente 225, Gávea
     Rio de Janeiro, BRAZIL   20000